"UNCLASSIFIED"

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)

2. REPORT DATE
1April98

3. REPORT TYPE AND DATES COVERED
Final 1May97 - 1Jan98

4. TITLE AND SUBTITLE
Computational Tools for Distributed Parameter Control Systems

5. FUNDING NUMBERS
F49620-97-C-0020

6. AUTHOR(S)
Dr. Christopher K. Allen
Dr. Gilmer Blankenship

AFRL-SR-BL-TR-98-
0282

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Techno-Sciences, Inc.
10001 Derekwood Lane, Suite 204
Lanham, MD 20706

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research/NM
110 Duncan Avenue, Rm B115
Bolling AFB
Washington, DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

19980414 070

13. ABSTRACT (*Maximum 200 words*)

The first phase of the design and development of a computer-aided design (CAD) tool for distributed parameter control systems has been finished. The kernel of the design tool has been designed and implemented for the two-dimensional situation. It is capable of modeling and simulating arbitrary systems described by partial differential equations. The implementation is based on a finite element method using fictitious domains and is written in C++. As such, the kernel can employ a regular grid for all problem geometries and simulate moving and/or deforming bodies without regridding.

We pay particular attention to the application of viscous, fluid-flow control. The design of an incompressible, viscous, fluid-flow simulator using the CAD kernel equations is presented in detail. Analytical work on the control flow problem is also presented. The work is based upon perturbation and asymptotic methods as well as analysis using differential geometry and Mathematica. The objective here is to determine analytic principles on which to base a feedback controller for flow control. The simulator would then be used to test the designs.

14. SUBJECT TERMS
Distributed parameter control, feedback control, viscous fluid flow, boundary layer control

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Techno-Sciences, Inc.

10001 Derekwood Lane
Suite 204
Lanham, Maryland 20706
(301)577-6000

Final Technical Report

Contract F49620-C-0020

Computational Tools for Distributed Parameter Control Systems

March 31, 1998

Dr. Christopher K. Allen
Principal Investigator

Abstract

The first phase of the design and development of a computer-aided design (CAD) tool for distributed parameter control systems has been finished. The kernel of the design tool has been designed and implemented for the two-dimensional situation. It is capable of modeling and simulating arbitrary systems described by partial differential equations. The implementation is based on a finite element method using fictitious domains and is written in C++. As such, the kernel can employ a regular grid for all problem geometries and simulate moving and/or deforming bodies without regridding.

We pay particular attention to the application of viscous, fluid-flow control. The design of an incompressible, viscous, fluid-flow simulator using the CAD kernel equations is presented in detail. Analytical work on the control flow problem is also presented. The work is based upon perturbation and asymptotic methods as well as analysis using differential geometry and Mathematica. The objective here is to determine analytic principles on which to base a feedback controller for flow control. The simulator would then be used to test the designs.

Table of Contents

Table of Figures

Final Technical Report

Computational Tools for Distributed Parameter Control Systems

1 Overview

Under the auspices of the U.S. government's Small Business Innovative Research (SBIR) program. Techno-Sciences, Inc. has contracted with the Air Force Office of Scientific Research (AFOSR) to design, develop and build computational software tools for the simulation and design of distributed parameter control systems (Contract F49620-C-0020). Particular emphasis was to be placed on applications of controlled fluid flow. The overall goal of the project is to develop a general-purpose Computer-Aided Design (CAD) tool capable of modeling and simulating a wide range of applications involving the control of distributed parameter systems. This document is a summary of the work performed in the Phase I effort.

We envisage the final product of this effort as a full-featured, stand-alone system with graphical front end similar to that of existing modeling software systems such as AutoCAD. Ultimately, the user would be able to model and simulate distributed parameter systems and design and simulate control subsystems for such systems. The CAD tool would allow the user to easily build complex systems from simpler subsystems by providing him or her with a palette of such subsystems from which to choose. Thus, the CAD tool represents the physical system as a modular set of interacting subcomponents, as well as providing simulation and control design capabilities.

In Phase I we used viscous fluid flows for our prototype of a distributed parameter system. We are currently considering a two-dimensional, incompressible, Newtonian fluid in arbitrary domains over arbitrary solid objects. The Navier-Stokes equations are used as the modeling equations in this situation. These equations are nonlinear and demonstrate chaotic behavior (turbulence). Moreover, they model certain physical behavior accurately enough to be of great practical interest. Thus, fluid flow control contains most of the features with which we are concerned when attempting to control a distributed parameter system.

We concentrated our efforts on the development of a general-purpose modeling environment for distributed parameter systems. In our paradigm, the objective system is comprised of a domain and a set of boundaries, some of which may be the surface of objects within the domain. We also assume that the system can be modeled by a set of partial differential equations with boundary conditions. The control actuators and sensors are assumed to be located on the boundaries in the domain. A set of C++ class libraries has been developed to model such situations. The class libraries implement a finite element/fictitious domain representation of a distributed parameter system. Once the target system is defined using the class library, it generates the discrete representation of the system suitable for numeric simulation on a computer. This representation is in a general format suitable for control system analysis. The control actuators may control the dependent variables or the geometry of the boundaries, while the sensors detect the state

of the dependent variables at the boundary surfaces. Initial tests of the modeling system were made for the particular case of two-dimensional Newtonian fluid flow.

Implementation of a finite element/fictitious domain discrete modeling system using object-oriented programming was a primary goal for us. This way at the very minimum the project would yield a very useful software library for the general-purpose modeling of mechanical and electromagnetic systems. We initially attempted to build the kernel by programming Matlab, however, this was found to be unsuitable for modeling the general situation (the programming was awkward and the problem setup times were unusually long). The advantage of this current software library over other finite element implementation is the use of the fictitious domain formulation and, also, the object-oriented framework. The object-oriented implementation makes the library modular and easy to use, the library being based on an intuitive model of the finite element. In fact, object-oriented programming is inherently designed to promote "reusability". The fictitious domain method is a special technique for handling the boundaries of the problem domain and their associated boundary conditions. This method is particular useful for treating complicated geometries and/or situations where the boundaries change with time. Thus, since we use a fictitious domain method for simulation, we have the option of actively modifying system geometries in order to achieve control.

Our initial control objectives were to actively control fluid flow around solid objects in order to suppress, or at least delay, the onset of turbulence. Intimately related to this goal is the reduction of drag forces on the object. We began studies on control of both the continuous Navier-Stokes equations and on the discrete (computer) approximations of these equations. As of yet we do not have any conclusive results concerning this topic. However, our preliminary findings do suggest general principles for the control of viscous fluids. In particular, we have performed a boundary layer analysis using perturbation methods in order to develop some physical principles on which to base our control strategy. The most notable findings there include the necessity of using multiple time-scales to describe transient behavior in boundary layers. Our current results for both the continuous and discrete control problems are included in the final section of this report.

The work done in Phase I may be divided into three categories:

- Design of CAD Kernel
- Viscous Fluid-Flow Simulator
- Control Problem Structure for Fluid Flow

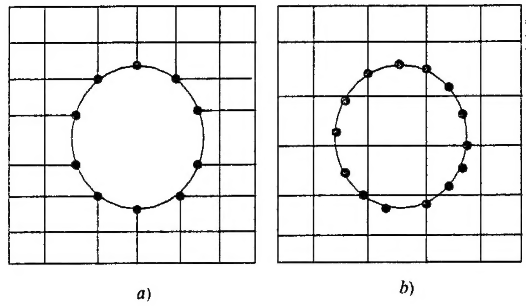
We cover each of these topics in a separate section of this report.

2 Design of the CAD Tool Kernel

2.1 Introduction

The kernel of the CAD tool is designed to model arbitrary distributed parameter systems which may be described with a set of Partial Differential Equations (PDEs). An initial attempt was made to build a prototype kernel using Matlab and a finite difference formulation. However, programming in Matlab was found to be much too cumbersome and awkward when attempting to model arbitrary systems. The current prototype kernel is written in C++, being based upon a class library we have written called FemLib. FemLib

implements a Finite Element Method (FEM) for computer modeling of distributed parameter systems; it is useful for any application using a finite element representation. The implementation of FemLib is based on a high degree of polymorphism over a small set of base classes. This allows the user to define an extremely broad range of systems and system topologies using a simple modeling paradigm. Once the objective system is described using finite elements, the class library will generate all the finite element matrices and tensors (including fictitious domain matrices - see below). These mathematical objects provide the discrete representation of the system which may then be manipulated using standard mathematical techniques. One may employ Matlab or M++ to do these computations. Small class libraries are built on top of FemLib to implement the specifics of the actual system under study. To simplify matters and reduce development time the Phase I implementation has been done for the two-dimensional (2D) situation. Therefore, all classes have been suffixed with the identifier "2D" to obviate this limitation. The three dimensional implementation would be directly analogous.



• Figure 1: FEM and FDM meshes

The FemLib class library is capable of modeling solid objects embedded in the problem domain using the Fictitious Domain Method (FDM), or Domain Embedding Method (DEM). This technique is typically used in conjunction with the finite element method. The general idea behind the fictitious domains is to model solid bodies in the problem domain (usually a connected subset of $\boldsymbol{R}^d$ where d=2,3) mathematically rather than using special meshes. By a special mesh we mean one that is required to contour around the object and, thus, the object is represented by the mesh itself. Accordingly, if the object at hand was to change or move, the mesh would have to be regenerated. In the fictitious domain method a regular mesh modeling the problem domain is ordinarily used. In addition to the domain mesh, boundary meshes are superimposed which represent the surfaces of embedded objects in the domain. The two types of meshes need not conform at all, see Figure 1 for the example of an infinite cylinder. In the mathematical formulation, dependent variables are introduced on the superimposed boundaries and are interpreted as Lagrange multipliers. These multipliers enforce the boundary conditions of the original problem.

The fictitious domain method makes the kernel implementation much easier and allows us to treat much more general problems. This is because the mesh used to model the domain and the mesh used to model the embedded objects are independent. This condition provides us with many important capabilities. It allows for the modular implementation of the CAD subsystem palette, system components described by a boundary mesh may be added or deleted easily. Also, we are able to model situations which would be very difficult to model, if not impossible, with the finite element method alone. For example, we may treat rigid body motion within a fluid without any regridding of domain. That is, the boundary mesh representing the rigid body may move through the domain mesh according to the equations of motion. Body deformations within a fluid may be treated just as easily. For example, compliant membranes and flexible structures may be modeled and simulated without expensive regridding. Also, we may perform shape

optimizations on the embedded objects to permit the maximization of certain design goals. Finally, we note that the FDM allows for the use of special fast solvers that exploit the regular structure of the domain mesh.

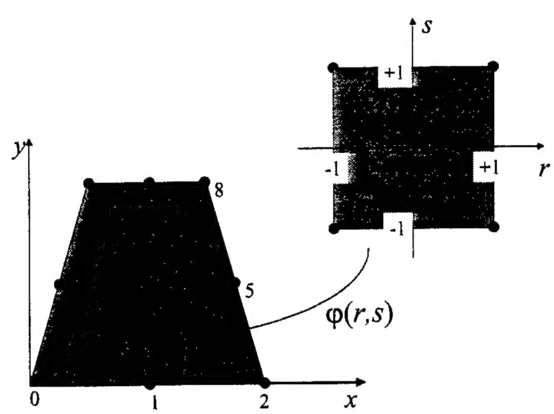
The major disadvantage of the fictitious domain method is the increased size in the discretized system. Since the boundaries are represented as additional meshes we must add the new node variables to the original system. Also, the fictitious domain method requires an additional coding effort not needed when using the FEM alone.

2.2 Finite Element Method

We implemented a standard finite-element approximation scheme for partial differential equations, such as that found in references [1], [2], and [3]. In this section the particulars of our implementation are covered to assist in the use of the FemLib C++ library. Due to the limited resources available for Phase I, the current version of FemLib models the two-dimensional situation only. Therefore, the discussion is limited to this case. For a detailed description of finite element methods see the above references.

2.2.1 The Elements

The basic idea behind the finite element method is to decompose the problem domain Ω into a number of smaller component regions, or finite elements. The solution of the partial differential system is then taken as a piecewise approximation across each of the finite elements. Within the element the field variable, say $u(\boldsymbol{x})$, is expressed as a linear combination of interpolation function known as shape functions $F_i(\boldsymbol{x})$, or basis functions, associated with the element. The set of all shape functions for all the elements of the domain forms a basis for a finite-dimensional vector space, which we denote $V_h(\Omega)$ (h is a parameter we include to emphasis the geometric size of the individual elements). It is this function space $V_h(\Omega)$ where we take the approximate solution to the partial differential system. The approximate solution will satisfy the integral, or weak, form of the partial differential equations where, typically, we use the shape functions themselves as the testing functions (Galerkin's method). Part of the art of finite elements is choosing the element shapes and the shape functions so that the function space $V_h(\Omega)$ contains a good approximation to the original problem.



• Figure 2: Finite element in grid and with local coordinates

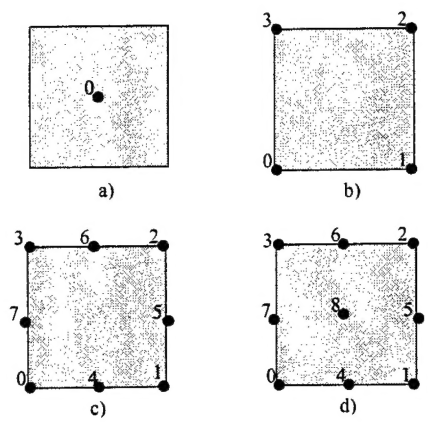
Although FemLib is setup to use any geometric shape, such as triangles or octagons, we shall concentrate solely on quadrilateral elements. This simplifies the discussion since the shape functions are easily described in the local coordinate system. Because we are

using the fictitious domain method, regular grids are typically used and this restriction presents no shortcoming to the modeling process. An example situation is shown in Figure 2. In the figure a trapezoidal region in Cartesian (x,y) space is decomposed into four quadrilateral elements. Also depicted in the figure is a single element shown in its local coordinate system. For the quadrilateral elements this is a Cartesian system for which we give the coordinates (r,s). Both r and s take values in the closed interval $[-1,1]$; that is, in the local coordinate system all the elements are squares! To convert back to the original problem domain, there exists for each element a mapping $\varphi(r,s)$ from the local (r,s) coordinates to the global (x,y) coordinates. This mapping is how the geometry of the system is captured in the discrete model. We shall see that the mapping is completely defined in terms of the vertices and the shape functions of the finite elements.

2.2.2 The Nodes

An important part of the finite element method is the use of node locations in the domain Ω. These are points in Ω usually having special significance with respect to the elements. For example, they tend to be vertices, centroids, and edge bisectors. This situation is depicted in Figure 2. There are nine nodes in the figure which are the vertices of the elements. Each node has an index i which is its index into the discrete system (in the figure the indices run from 0 to 8). This index does not change throughout the simulation. (Each node may have multiple element indices, however. See the discussion concerning Figure 3.) Thus, it is possible to change the structure of the finite element matrices simply by relabeling the nodes' system indices. Note that the nodes may be shared by more that one element, this is how the topology of the system is captured in the discrete system; it results in coupling between the nodes.

The number and position of the nodes on an element depends upon the interpolation scheme used in the simulation. Typically, a polynomial approximation is used and, therefore, only the order of the approximation is variable. Figure 3 shows the node locations for the four interpolation schemes implemented in FemLib. Note that as the order of approximation increases the number of nodes in the element must increase. Also shown in the figure are the element indices n for the nodes. Each finite element maintains a dynamic array of nodes that it contains and the element index is the means by which it references its nodes. This index is different from the more ubiquitous system index in that each node may have a different element index depending upon which element is referencing it. The node locations, as indicated in the figure, have local coordinate values of either −1, 0, or +1. For example, nodes 0, 8, and 2 of the biquadratic element in Figure 3 d) have (r,s) coordinates (−1, −1), (0,0), and (+1,+1), respectively.



• Figure 3: Node positions and indexes: a) Constant, b) Bilinear, c) Biquadratic serendipity, and d) Biquadratic

2.2.3 The Shape Functions

Here we list the shape function for each different element. Defining r_n and s_n as

$$\begin{aligned} r_n &\equiv \{-1,+1,+1,-1\} \\ s_n &\equiv \{-1,-1,+1,+1\} \end{aligned} \qquad (1)$$

we have the following:

For the constant element

$$F_0(r,s) = \begin{cases} 1 & \text{if } (r,s) \in [-1,+1]\times[-1,+1] \\ 0 & \text{if } (r,s) \notin [-1,+1]\times[-1,+1] \end{cases} \qquad (2)$$

For the bilinear element

$$F_n(r,s) = \frac{1}{4}(1 - r_n r)(1 - s_n s) \quad \text{for } n = 0,\ldots,3 \qquad (3)$$

For the biquadratic serendipity

$$\begin{aligned} F_n(r,s) &= \frac{1}{4}(1 - r_n r)(1 - s_n s)(r_n r + s_n s - 1) \quad \text{for } n = 0,\ldots,3 \\ F_4(r,s) &= \frac{1}{2}(1 - r^2)(1 - s) \\ F_5(r,s) &= \frac{1}{2}(1 + r)(1 - s^2) \\ F_6(r,s) &= \frac{1}{2}(1 - r^2)(1 + s) \\ F_7(r,s) &= \frac{1}{2}(1 - r)(1 - s^2) \end{aligned} \qquad (4)$$

For the full biquadratic

$$\begin{aligned} F_n(r,s) &= \frac{1}{4}(1 - r_n r)(1 - s_n s) r_n r s_n s \quad \text{for } n = 0,\ldots,3 \\ F_4(r,s) &= -\frac{1}{2}(1 - r^2)(1 - s) s \\ F_5(r,s) &= +\frac{1}{2}(1 + r)(1 - s^2) r \\ F_6(r,s) &= +\frac{1}{2}(1 - r^2)(1 + s) s \\ F_7(r,s) &= -\frac{1}{2}(1 - r)(1 - s^2) r \\ F_8(r,s) &= -\frac{1}{2}(1 - r^2)(1 - s^2) \end{aligned} \qquad (5)$$

Note that each shape function F_n is unity at the node n and zero at all the other nodes on the element. That is, in local coordinates we have the following relation:

$$F_n(r_m, s_m) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases} \qquad (6)$$

where (r_m, s_m) are the local coordinates of element node m. In the finite element method the values of field variable u are computed only at the discrete node locations. Thus, the shape functions provide the interpolation of u between these node locations.

2.2.4 Integration

The finite element method, as opposed to the finite difference method, is based upon integration. Therefore, we need to know how to integrate functions over the elements, this is where the mapping φ comes into play. Recall from Figure 2 that φ maps the local element coordinates (r,s) to the global coordinates (x,y) in the domain where the element is located. It can be varified that this mapping is given by

$$(x,y) = \varphi(r,s) \equiv \sum_n (x_n, y_n) F_n(r,s) \qquad (7)$$

where the summation is taken over all the nodes n of the element and the pairs (x_n, y_n) are the global coordinates of the n^{th} node. Say we wish to compute the integral of some function $f(x,y)$ over some finite element T. From the calculus we know that this can be done in the local coordinates (r,s) of T using the change of variable formula

$$\iint_T f(x,y)\,dxdy = \int_{-1}^{+1}\int_{-1}^{+1} f(\varphi(r,s)) \frac{\partial\varphi(r,s)}{\partial(r,s)}\,drds \qquad (8)$$

where $\partial\varphi(r,s)/\partial(r,s)$ is the Jacobian of the transform $\varphi(r,s)$. In FemLib, all the finite element integrals are computed using this formula. The integrals are computed numerically using various quadrature schemes that may be selected by the user (e.g., Simpson, Bode, Gaussian).

In practice, the shape functions are actually associated with the nodes of the system and are indexed by the system index i. Thus, the shape function F_i is defined piecewise over each element that contains node i. The shape function used on each element depends upon the particular element index n of the node, however, the system is setup in such a way that the shape functions are continuous. Note also that each shape function F_i has compact support since it is nonzero only on the elements that contain node i. In this context the set of shape functions $\{F_i(x,y)\}$ for a basis for vector space, the vector space of approximation functions. Since

$$F_i(x_j, y_j) = \begin{cases} 1 & \text{for } j = i \\ 0 & \text{for } j \neq i \end{cases} \qquad (9)$$

where $(x_j, y_j) \in \mathbf{R}^2$ is the coordinate position of node j, any approximation of the field variable $u(x,y)$ is expanded in terms of these basis functions according to

$$u(x,y) \approx \sum u_i F_i(x,y) \qquad (10)$$

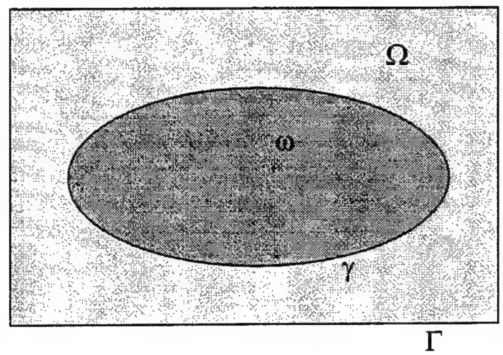
where the u_i are (the expansion coefficients) the values of $u(x,y)$ at the node locations (x_i, y_i). It is the goal of the finite element method to find the set coefficients $\{u_i\}$.

2.3 Fictitious Domain Method

To see the main idea of the fictitious domain method, consider the example of a classical Dirichlet problem, as in reference [4]. The problem is

$$\begin{aligned} \alpha u + \nu\nabla^2 u &= f \text{ in}\omega \\ u &= g \text{ on}\gamma \end{aligned} \qquad (11)$$

where α and ν are positive real parameters, u is the dependent variable, and f represents external forces. The boundary conditions are given by the function g which is prescribed along the boundary of the domain $\gamma=\partial\omega$. This situation is depicted graphically in Figure 4 where the original domain ω is embedded in a larger domain Ω whose boundary is Γ.



• Figure 4: Dirichlet Problem Domain and Boundary

2.3.1 The Variational Formulation

In the spirit of finite element method, we want to express the above problem in a variational, or weak, form. This is done by multiplying the original equation by a testing function $v(x)$ then integrating the Laplacian term by parts. Modulo technical details, the resulting equation should hold for any such testing function v. By properly choosing the set of allowed testing functions we can eliminate the boundary term arising from the integration by parts to secure the following description:

Find u in V_g such that

$$\int_\Omega [\alpha uv + \nu\nabla u \cdot \nabla v] d\Omega = \int_\Omega fv d\Omega \qquad \forall v \in V_0 \qquad (12)$$

where V_g and V_0 are the function subspaces given by

$$\begin{aligned} V_g &= \{u \in H^1(\omega) | u = g \text{ on}\gamma\}, \\ V_0 &= H_0^1(\omega) = \{u \in H^1(\omega) | u = 0 \text{ on}\gamma\}. \end{aligned} \qquad (13)$$

This is the problem that the standard finite element method approximates. The field variable u is expanded in terms of the shape functions, according to Eq. (10), where the node values u_i along the boundary γ are given by g (either directly or with some sort of integral averaging). This ensures that the approximation for u is in the set V_g. Therefore,

we need only determine the value of u_i for each internal node of the domain ω. Note that the set of shape functions associated with these internal nodes form a basis for a finite-dimensional space V_0^h contained in V_0. Thus, these shape functions form a set of independent testing functions, one for each internal node. Applying Eq. (12) to each of these functions yields a linear system of N equation in N unknowns, where N is the number of internal nodes in ω. That system may then be solved using standard numeric techniques to find the values of u_i for the internal nodes and, thus, the full approximation.

2.3.2 The Fictitious Domain Formulation

Solving the variational form using the finite element method requires that the domain ω and boundary γ are approximated by the finite element grid. If the boundary is complicated this may be a difficult, or expansive task. Or, if the boundary changes with time the domain must be regridded at each time step. The fictitious domain method sidesteps this issue by embedding ω in a larger, simpler, domain Ω then enforcing the boundary conditions on γ mathematically.

The fictitious domain method may be formalized using the Lagrangian for the original system. This procedure, in effect, results in a minimum energy argument. The formal Lagrangian L for Eq. (11) is given by the following:

$$L(u) \equiv \frac{1}{2}\int_{\omega}\left[\alpha u^2 + v|\nabla u|^2\right]d\omega - \int_{\omega} fu\,d\omega \;\; \forall u \in V_g \qquad (14)$$

It is well known that minimizing the above Lagrangian with respect to u in V_g yields the weak solution to the Dirichlet problem (indeed, the necessary condition $dL/du=0$ produces the variational formulation for the problem). Note, however, that this minimization may be interpreted as a constrained minimization due to the requirements of the set V_g, specifically, u must equal g on the boundary γ.

Using the Lagrange multiplier theorem we may reformulate the weak problem as the following unconstrained minimization

$$\min_{u}\max_{\lambda} \;\; L(u,\lambda) = \int_{\omega}\left[\alpha u^2 + v|\nabla u|^2\right]d\omega - \int_{\omega} fu\,d\omega + \int_{\gamma}\lambda(u-g)\,d\gamma \quad \forall u \in V, \lambda \in H^{-1/2}(\gamma) \qquad (15)$$

where this time the solution space V may be chosen much more arbitrarily. (We take $\lambda \in H^{-1/2}(\gamma)$, the dual of the trace space $H^{1/2}(\gamma)$, simply out of convenience. In actuality we assume that λ is in the larger space $L^2(\gamma)$.) In fact, we shall choose V so as to expand the actual problem domain to that of Ω. Taking the functional derivative of $L(u,\lambda)$ with respect to both u and λ yields the variational formulation for the "fictitious domain" Dirichlet problem

Find u in V and λ in $H^{1/2}(\gamma)$ such that

$$\begin{aligned} &\int_{\Omega}\left[\alpha uv + v\nabla u \cdot \nabla v\right]d\Omega = \int_{\Omega} fv\,d\Omega + \int_{\gamma}\lambda v\,d\gamma && \forall v \in V \\ &\int_{\gamma}\mu(u-g)\,d\gamma = 0 && \forall \mu \in H^{-1/2}(\gamma) \end{aligned} \qquad (16)$$

where we take

$V = H_0^1(\Omega)$. (17)

After finding the solution u to this problem on the larger domain Ω, the restriction $u|_\omega$ will be the weak solution to the original Dirichlet problem.

The boundary conditions are enforced in Eq. (16) by the term involving the Lagrange multiplier λ. We must solve for the new variable λ, as well as for the dependent variable u, in order to solve the entire system. Therefore, mathematically the situation is somewhat more complicated but this presents no real problem when solving the system numerically. By comparing the two terms on the RHS of the first of Eqs. (16) we see that the Lagrange multiplier λ appears as a fictitious external force, or impulsive forcing function, acting along the boundary γ. This force is exactly that which would be caused by the original boundary conditions acting on u. For our current system that force is

$$\lambda = \nu \frac{\partial u}{\partial \mathbf{n}}, \qquad (18)$$

where $\mathbf{n}$ is the unit normal to the boundary surface. This may be shown using Green's theorem applied to the variational formulation in (16).

Note that fictitious domain formulation is readily solved (numerically) using a finite element approach. Indeed, one may choose the auxiliary domain Ω as a regular subset of $\mathbf{R}^d$ that conforms to the coordinates (e.g., a coordinate rectangle for cartesian coordinates). We may then divide Ω into regular, homogeneous finite elements so that the resulting matrices will demonstrate a high degree of structure and symmetry.

2.4 Computer Implementation

Here we describe the computer software architecture for implementing the CAD tool. An initial investigation using a finite differencing technique programmed in Matlab was attempted early in the project. It was found that the performance of Matlab was prohibitively slow when doing the setup calculations (computer modeling) and successive over-relaxations for the time-stepping solutions. Formulating the programming paradigm for system modeling was also very clumsy in the Matlab programming language. Therefore we have implemented the problem formulation in the C++ programming language using the M++ class library for matrix operations. From the C++ code we can create MEX files for use with Matlab. In this manner the resulting system matrices can be send to Matlab to do the command, numeric, and graphical processing. Since matrix operations and manipulations in Matlab are relatively fast, and we have a large variety of visualization tools available for viewing the results, this seems to be a reasonable setup.

We also describe here some of the finite element/fictitious domain modeling strategy since it is relevant to the implementation and to the structure of the control problem. For example, in order to circumvent the well-known divergence instability condition, which causes wild pressure fluctuations in the numeric integration of the Navier-Stokes equations, we provide the capability of using two different grids for the pressure and velocity variables. Thus, the implementation allows for different sets of basis function expansions on the same grid. This leads to different (discrete) system structure.

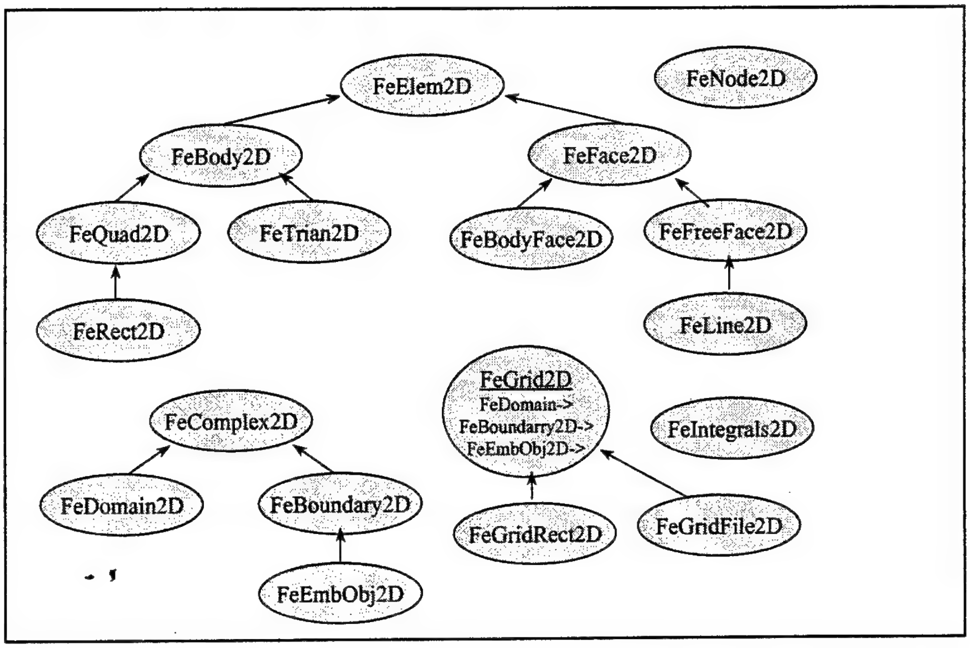
2.4.1 The Finite Element/Fictitious Domain Library (FemLib)

The computer modeling portion of the project has been implemented as a stand alone, general purpose set of C++ class libraries. The most basic library is the finite element library, called FemLib, that implements all the requirements of a finite element/fictitious domain formulation. This library is useful for any application using a finite element representation of a distributed parameter system. The implementation is based on a high degree of polymorphism over a small set of basic classes. This allows the user to define an extremely broad range of systems and system topologies using a simple modeling paradigm. Once the objective system is described using finite elements, the class library will generate all the finite element (including fictitious domain) matrices and tensors with the M++ class library representation. These mathematical objects provide the discrete representation of the system which may then be manipulated using standard mathematical techniques. One may employ Matlab or M++ itself to do these computations. The other class library on which we shall focus is the Navier-Stokes class library. This is a small set of classes that are used when the governing system equations are the Navier-Stokes equations.

To simplify matters and reduce development time the Phase I implementation has been done for the two-dimensional (2D) situation. Therefore, all classes have been suffixed with the identifier "2D" to obviate this condition. The three dimensional implementation would be directly analogous.

At the heart of the finite element implementation are the virtual base classes *FeElem2D, FeComplex2D*, and *FeGrid2D*. Also important is the stand-alone class *FeNode2D*. A class hierarchy diagram is shown in Figure 5. The standard convention is to represent inheritance via a directed arrow from the derived class (subclass) to the base class (superclass). From this diagram we see that the class *FeElem2D* is the base for all finite element classes. As it stands, it is simply a container of *FeNode2D* objects, which represent the nodes within the finite element. From there the derived classes add the various attributes and features particular to any finite element.

The *FeComplex2D* class is a container class for *FeElem2D* objects. This makes sense since a finite element grid is simply a collection of finite elements that fit together "nicely" (a complex) such that the topology is easily inferred. The class *FeComplex2D* is a base for two types of sub-complexes, one defining the problem domain, *FeDomain2D*, and one defining the problem boundaries, *FeBoundary2D*. Also, *FeBoundary2D* has the derived class *FeEmbObj2D* representing embedded objects in the fluid domain. Therefore, in the current two-dimensional implementation *FeDomain2D* contains two-dimensional finite elements (derived from *FeBody2D*) and *FeBoundary2D* contains one-dimensional finite elements (derived from *FeFace2D*). Finally, the complete finite element grid is represented by the *FeGrid2D* base class. This class has three major attributes. The first is a *FeDomain2D* object representing the domain of the grid and the second is a container of *FeBoundary2D* objects which represent all the boundaries in the grid's domain. In this manner we may associate a separate boundary object for each different boundary condition. The third is a container of embedded objects. A separate *FeEmbObj2D* object is contained for each embedded solid in the domain.



• Figure 5: Finite Element Class Library Inheritance Diagram

Notice that there are two base classes of *FeElem2D*, they are *FeBody2D* and *FeFace2D*. The *FeBody2D* class designates finite elements containing area (volume for the 3D situation) and representing the actual component bodies from which the grid is composed. The *FeFace2D* class is meant to represent components of boundaries within the domain. Since there are two types of boundaries in the domain that we wish to consider, element body faces and free boundaries (i.e., boundaries of embedded objects), there are two subclasses derived from *FeFace2D*. The first, *FeBodyFace2D*, represents the face boundary of a *FeBody2D* object while the second, *FeFreeFace2D*, represents an arbitrary boundary in the domain. This situation is natural because the shape functions of the body faces are dependent upon the interpolation scheme used in the body element. The interpolation scheme and, therefore, the shape functions for a free boundary are arbitrary. However, we are currently revising this implementation so that the *FeFace2D* objects naturally know whether or not they are body faces. When completed the new implementation will eliminate the need for the two derived classes.

We consider the *FeElem2D* hierarchy in more detail. Figure 6 shows code excerpts from these finite element classes. In the figure the most important attributes and member of each of the base classes are listed. (The type *Cardinal* represents cardinal numbers, the type *Real* represents

```
FeElem2D
    Cardinal        nNodes;     // no. of nodes
    Cardinal        nVertices;  // no. of vertices
    ArrayOfR2s      arrVertices; // element vertices
    ArrayOfFeNodes arrNodes;    // element nodes

FeBody2D
    Interpolation {Const, BiLinear, BiQuadratic, ...};
    virtual Real  ShapeFunc(Index iNode, R2 pt) = 0;
    virtual R2    ShapeGrad(Index iNode, R2 pt) = 0;

FeFace2D
    Interpolation {Const, Linear, Quadratic, ...};
    virtual Real  ShapeFunc(Index iNode, Real pt) = 0;
    virtual R2    ShapeGrad(Index iNode, Real pt) = 0;
```

• Figure 6: Attributes of the Finite Element Classes

elements of the set of real numbers $\boldsymbol{R}$, while the type (class) *R2* represents elements of $\boldsymbol{R}^2$.) The *FeElem2D* class consists primarily of an array of vertices (points in $\boldsymbol{R}^2$) and an array of *FeNode2D* objects. The objects are contained in arrays since the vertices and nodes of a finite element are indexed according to the type of element. The *FeBody2D* and *FeFace2D* classes define an additional attribute, the enumeration *Interpolation*. This attribute determines the type of interpolation scheme used by the element, specifically, the degree of polynomial representing the shape function. The interpolation scheme for each element can be changed by simply setting the appropriate flag. Two member function are also declared in each class, *ShapeFunc*() and *ShapeGrad*(). These functions represent, respectively, the shape function and the its gradient for the finite element. The signatures of the functions in each of the two classes are different since the local coordinates of a (2D) body are two-dimensional while the local coordinates of a (2D) face are one-dimensional. The two functions are declared as pure virtual functions and must be defined in any instantiable base class, since a finite element without a shape function is useless. The functions are meant as placeholders so one can access the shape function and its gradient without knowing the particulars of the finite element itself.

Finally, the classes *FeQuad2D*, *FeRect2D*, *FeTrian2D*, *FeLine2D* represent actual, instantiable finite elements. The class *FeQuad2D* represents an arbitrary quadrilateral body. The *FeRect2D* represents a rectangular body and is derived from *FeQuad2D* (since a rectangle is a quadrilateral). Due to the regular structure of a rectangle some of the coordinate functions and shape functions are faster to compute than with a *FeQuad2D*. The *FeTrian2D* class implements an arbitrary triangular body while the *FeLine2D* represents a boundary element comprised of a line segment.

Now consider the *FeGrid2D* hierarchy. As we have mentioned, the base class *FeGrid2D* contains three major attributes, a *FeDomain2D* object, a list of *FeBoundary2D* objects, and a list of *FeEmbObj2D* objects. The topologies of these contained objects are determined by the sharing of nodes between the finite elements. That is, the finite elements recognize their neighboring elements by sharing common nodes, typically at vertices. The *FeGrid2D* class is simply a container class of finite element objects and no provisions are made to create any such grids. That task is accomplished through the use of derived classes. There are currently two derived classes of *FeGrid2D*, the class *FeGridRect2D* which implements a rectangular grid and *FeGridFile2D* that implements a grid defined by a formatted input file (currently unfinished). Since we are using rectangular grids in the test bed problems it was convenient to provide a special implementation for that situation. For more arbitrary situations the user can define a grid with an ASCII file containing the nodes and elements, and their positions and connections.

The last class we mention in the finite element class library is *FeIntegrate2D*. This class computes the discrete system data from *FeGrid2D* objects. That is, it computes the matrices and tensors common to a Galerkin-type finite element/fictitious domain approximation of the model problem. The discrete representations of the bilinear and trilinear forms appearing in the variational formulation are computed in terms of the basis of shape functions over the elements. For example, the standard inner product of functions over the grid space has matrix entries (the basis matrix) given by

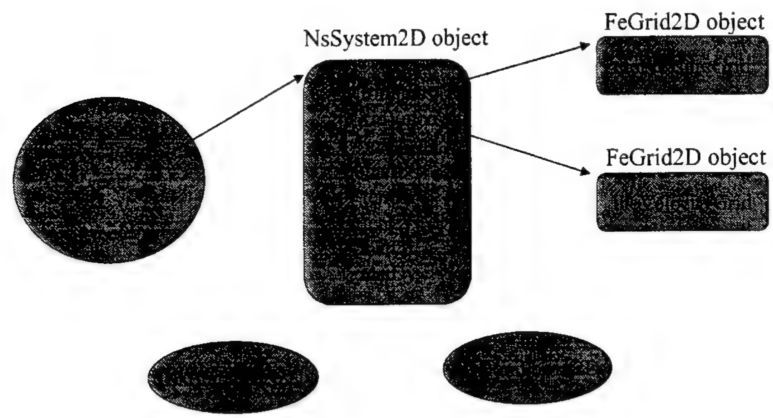
$$A_{ij} = \int_{\Omega} F_i F_j \, d\Omega \qquad (19)$$

where F_i is the shape function for the (internal) node i, while the $H^1(\Omega)$ inner product has matrix entries (the stiffness matrix) are given by

$$C_{ij} = \int_{\Omega} \nabla F_i \cdot \nabla F_j \, d\Omega . \qquad (20)$$

2.4.2 The Navier-Stokes Class Library (FluidLib)

The viscous fluid-flow simulator is implemented as a small C++ class library, FluidLib, which sits on top of FemLib. FluidLib currently realizes the finite element/fictitious domain formulation of the 2D incompressible Navier-Stokes equations. It also provides for the simulation of boundary sensors and actuators for a feedback control system.



• Figure 7: FluidLib classes

The simulator functions by first retrieving from FemLib the matrices and tensors representing the linear, bilinear, and trilinear forms specific to the integral formulation of the Navier-Stokes equations (we use a primitive variable formulation). These matrices and tensors are assembled into a nonlinear, discrete system that approximates the time evolution of a viscous, incompressible fluid. These equations are integrated in time using a three-step "θ-scheme". This method is a generalization of the Peaceman-Rachford scheme and was introduced by Glowinski in the mid 1980's.

Currently, there are four classes in FluidLib, *NsBndForms2D*, *NsDomForms2D*, *NsSystem2D*, and *NsTimeStep2D*. The relationships between these classes are shown in Figure 7. In the figure, an oval represents a class where a rounded rectangle represents an actual object of an instantiable class. The classes *NsBndForms2D* and *NsDomForms2D* are essentially helper classes that compute the bilinear and trilinear forms associated with the weak form of the Navier-Stokes equations. The class *NsSystem2D* actually holds these forms, represented as matrices and rank-three tensors (these data are stored as M++ matrices and arrays, respectively). *NsSystem2D* also owns the pressure and velocity grid (*FeGrid2D*) objects. The time integration of the Navier-Stokes equations is done in the class *NsTimeStep2D*. This class owns the *NsSystem2D* object and maintains the dependent variables (i.e., the velocities and pressure). As mentioned above, this class implements the θ-scheme for numerical integration. For the mathematical details of the fluid-flow simulator see the next section.

3 Viscous Fluid-Flow Simulator

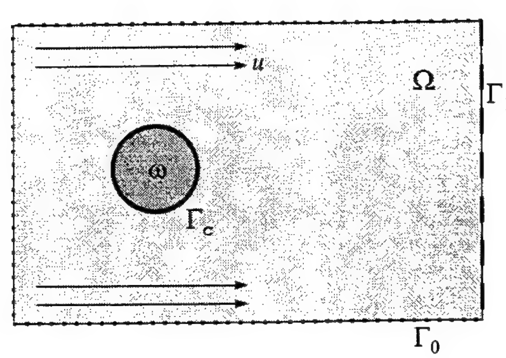
Here we describe the techniques used to build the viscous, fluid-flow simulator. Most of the material discussed is implemented in the C++ library FluidLib. The presentation is rather technical since we discuss the details encountered in the implementation. There would be no loss of continuity if the reader were to proceed directly to Section 4.

3.1 Mathematical Formulation

The governing equations are the incompressible Navier-Stokes equations. The situation we describe is uniform flow around a solid object embedded in the fluid domain. The situation is described mathematically by the following partial differential system:

$$
\begin{array}{ll}
\dot{\mathbf{U}} + (\nabla \cdot \mathbf{U})\mathbf{U} + \nabla p - \nu \nabla^2 \mathbf{U} = \mathbf{f} & \text{in}\,\Omega \\
\nabla \cdot \mathbf{U}(t) = 0 & \text{in}\,\Omega \\
\mathbf{U}(t) = \mathbf{U}_{\text{inf}} & \text{on}\,\Gamma_0 \\
p(t)\mathbf{n} - \dfrac{\partial \mathbf{U}(t)}{\partial \mathbf{n}} = 0 & \text{on}\,\Gamma_1 \\
\mathbf{U}(t) = \mathbf{c}(t) & \text{on}\,\Gamma_c
\end{array}
\qquad (21)
$$

where $\mathbf{U}(t) \in \mathbf{R}^d$ (d=2,3) is the velocity vector, $p(t) \in \mathbf{R}$ is the pressure (normalized to fluid density), $\nu \in \mathbf{R}_+$ is the viscosity, $\mathbf{f}(t) \in \mathbf{R}^d$ is the external force on the fluid, $\mathbf{U}_{inf} \in \mathbf{R}^d$ is the fluid velocity at infinity (i.e., uniform flow velocity), $\mathbf{c}(t) \in \mathbf{R}^d$ is the control, $\mathbf{n} \in \mathbf{R}^d$ is the normal vector to the respective boundary, Ω is the problem domain, Γ_0 is the Dirichlet boundary, Γ_1 is the free-flow boundary,



• Figure 8: Fluid flow problem

and Γ_c is the control boundary (boundary of the embedded solid object). See Figure 8 for a graphical depiction of this situation. Note that Γ_c might be a function of time.

The function space of solutions for velocity $\mathbf{U}(t)$ will be denoted $V(\Omega)$ (or simply V) while the solution space for pressure $p(t)$ will be $P(\Omega)$ (or simply P). We will also consider (vector-valued) functions spaces on the boundaries, denoted by $B(\Gamma_0)$ and $B(\Gamma_c)$. Note that $V(\Omega)$ should be a subspace of the Sobelov space $H^1(\Omega)$. For convenience of reference, Table 1 collects the definitions of these spaces and other function spaces used in this formulation.

Now consider the weak, or "variational", formulation for the system. Using a fictitious domain formulation (see Section 2.3.2) for the internal boundary we have

$$
\begin{aligned}
&\int_\Omega \dot{\mathbf{U}} \cdot \mathbf{V} d\Omega + \int_\Omega (\mathbf{U} \cdot \nabla)\mathbf{U} \cdot \mathbf{V} d\Omega - \int_\Omega p \nabla \cdot \mathbf{V} d\Omega + \nu \int_\Omega \nabla \mathbf{U} \cdot \nabla \mathbf{V} d\Omega = \\
&\qquad \int_\Omega \mathbf{f} \cdot \mathbf{V} d\Omega + \int_{\Gamma_c} \Lambda \cdot \mathbf{V} d\Gamma \qquad\qquad \forall \mathbf{V} \in V_0(\Omega) \\
&\int_\Omega \nabla \cdot \mathbf{U} q d\Omega = 0 \qquad\qquad \forall q \in P_1(\Omega) \qquad (22) \\
&\int_{\Gamma_0} \mathbf{U} \cdot \mathbf{b} d\Gamma = \int_{\Gamma_0} \mathbf{U}_{\text{inf}} \cdot \mathbf{b} d\Gamma \qquad\qquad \forall \mathbf{b} \in B(\Gamma_0) \\
&\int_{\Gamma_c} \mathbf{U} \cdot \mathbf{b} d\Gamma = \int_{\Gamma_c} \mathbf{c} \cdot \mathbf{b} d\Gamma \qquad\qquad \forall \mathbf{b} \in B(\Gamma_c)
\end{aligned}
$$

Note that the boundary conditions on Γ_1 are absorbed naturally into the weak formulation (this is by design). Referring to the first line in the above, the "testing" functions $\mathbf{V}$ live in $V_0(\Omega)$, which is the subspace of $V(\Omega)$ given by

$$V_0(\Omega) = \left\{ \mathbf{V} \in V(\Omega) \middle| \mathbf{V}|_{\Gamma_0} = 0 \right\} \qquad (23)$$

That is, $V_0(\Omega)$ is the subspace of functions in $V(\Omega)$ which are zero along the boundary Γ_0. A final point with respect to the first line in (22) is that the vector function Λ is the Lagrange multiplier resulting from the fictitious domain formulation. It enforces the internal boundary conditions on Γ_c. However, this function is now an additional dependent variable for which to solve.

In the second line of (22) we take our testing functions from the subspace $P_1(\Omega) \subset P(\Omega)$. This is the subspace of pressure functions that are zero along the boundary Γ_1 (rather than Γ_0) or, more precisely,

$$P_1(\Omega) \equiv \left\{ q \in P(\Omega) \middle| q|_{\Gamma_1} = 0 \right\} \qquad (24)$$

The motivation for this particular choice of test functions will become clear later. The function spaces are summarized in the following table:

Space	Description
$V(\Omega)$	Solution space for $\mathbf{U}(t)$, generally a subspace of Sobolev space $[H^1(\Omega)]^d$
$V_0(\Omega)$	Space of velocity testing functions, elements of $V(\Omega)$ which are 0 on Γ_0
$P(\Omega)$	Solution space for $p(t)$, generally a subspace of Sobolev space $H^1(\Omega)$
$P_1(\Omega)$	Space of pressure testing functions, elements of $P(\Omega)$ which are 0 on Γ_1
$B(\Gamma_0)$	Space of boundary condition testing functions, subspace of $[H^{1/2}(\Gamma_0)]^d$
$B(\Gamma_c)$	Solution and testing space for $\Lambda(t)$, subspace of Sobolev space $[H^{1/2}(\Gamma_c)]^d$

Table 1: Function spaces for Navier-Stokes fictitious domain formulation

3.2 System Discretization - Finite Elements and Fictitious Domains

For the discrete approximation we use a finite element approach. Therefore, a finite triangulation $\Im_h$ of Ω is assumed (the "triangulation" does not necessarily consist of triangles, just some collection of polytopes). This triangulation depends upon the parameter *h*, the triangulation becoming increasingly finer as *h* is approaches zero. Next a function space V^h is formed such that each element therein may be represented by a polynomial of some finite degree *n* over each triangle *T* in the triangulation $\Im_h$. This condition may be written more precisely as

$$V^h(\Omega) = \left\{ v_h \in V(\Omega) \middle| v_h|_T \in Poly(n), \forall T \in \Im_h \right\} \qquad (25)$$

where *Poly(n)* is the space of polynomials over Ω with degree less than or equal to *n*. Also associated with the triangulation is a set of node locations $\{x_j\}^h$ on the domain Ω. These nodes are the sample points for the dependent variables, that is, their values are defined at these discrete locations.

Incidentally, the bulk of the programming effort lies in the triangulation of the domain Ω and the representation of the function spaces over the triangulation. Once the triangulation and function spaces have been defined, the resulting discrete system may be attacked using existing software, such as Matlab or M++.

3.2.1 Shape Functions

Obviously $V^h(\Omega)$ has a finite basis. The basis we shall consider is the set of standard shape functions $\{F_i(x)\}$ for finite elements in the triangulation. These shape functions have the following properties

$$F_i(x_j) = \begin{cases} 1 & \text{for } j = i \\ 0 & \text{for } j \neq i \end{cases} \qquad (26)$$

where $x_j \in \mathbf{R}^d$ is the coordinate position of node *j*. Moreover, $F_i(x)$ has support only on the finite elements $T \in \Im_h$ to which node *i* is common. In other words, $F_i(x)$ is nonzero only on the elements $T \in \Im_h$ in which point x_i is contained. We shall denote the set of triangles containing xi as $\Im_h(i)$ so this property may be formally written

$$F_i(x) = 0 \quad \forall x \in \Im_h \setminus \Im_h(i) \qquad (27)$$

Thus, we see that the shape functions provide an interpolation scheme between nodes. Note that we may have different sets of shape functions for the same triangulation depending upon the degree *n* of polynomial space $V^h(\Omega)$.

Since the shape functions form a basis for $V^h(\Omega)$ we may approximate the dependent variable $\mathbf{U} \in V^h(\Omega)$ as a finite expansion of these functions. Henceforth, for concreteness we assume that the fluid domain Ω is two dimensional and we separate the vector $\mathbf{U}=(u,v)$ into its scalar components to save storage space in the resulting discrete system. Then the basis function expansion for $\mathbf{U}$ is

$$\mathbf{U} = \begin{pmatrix} \sum_i u_i \, F_i(x) \\ \sum_j v_j \, F_j(x) \end{pmatrix}. \qquad (28)$$

Likewise, we may invoke a similar procedure for the other dependent variables. Letting $\{G_i(x)\}$ be the set of shape functions for the pressure approximation space $P^h(\Omega)$ and $\{H_i(x)\}$ be the set of shape functions for the boundary function approximation space $B^h(\Gamma_c)$, we have

$$p = \sum_i p_i \, G_i(x) \qquad (29)$$

and

$$\Lambda = \begin{pmatrix} \sum_i \lambda_i \, H_i(x) \\ \sum_j \mu_j \, H_j(x) \end{pmatrix} \qquad (30)$$

where $\Lambda=(\lambda,\mu)$.

3.2.2 Index Sets

Inserting the above expansions into the weak formulation (22) and using the shape functions themselves as the testing functions (yielding Galerkin's method) produces the system of discrete matrix-vector (and tensor) equations. Before we do this, however, some indexing sets are introduced. For the set of sample nodes $\{x_i\}^h$ on the function space $V^h(\Omega)$ let

$$I \equiv \{\, 0,1,2, \ldots, N_V - 1 \,\} \qquad (31)$$

be their index set, where N_V is the number of nodes in $\{x_i\}^h$. Now let $I_0 \subseteq I$ be the set

$$I_0 \equiv \left\{ i \in I \,|\, F_i(x)\big|_{\Gamma_0} = 0 \right\} \qquad (32)$$

which is the set indices for the "internal nodes" not on the Dirichlet boundary Γ_0. Thus,

$$V_0^h(\Omega) = \mathrm{span}\{F_i(x) | i \in I_0\}. \qquad (33)$$

(The space $V_0^h(\Omega)$ is the approximation of $V_0(\Omega)$). We also define the set of nodes indices on the boundary Γ_0 as

$$I_B \equiv I \setminus I_0 . \qquad (34)$$

Regarding the nodes of the other approximation spaces, we define their index sets in a similar fashion. The index set for $\{G_i(x)\}$ is given by

$$J = \{\, 0,1,2, \ldots, N_P - 1 \,\} \qquad (35)$$

where N_P is the number of nodes in the space $P^h(\Omega)$. The index subset for $P_1^h(\Omega)$ is given by

$$J_1 \equiv \left\{ i \in J \,|\, G_i(x)\big|_{\Gamma_1} = 0 \right\} \qquad (36)$$

which is the set of "internal pressure nodes" not on the free-flow boundary Γ_1. Thus,

$$P_1^h(\Omega) = \text{span}\left\{ G_i(x) \,|\, i \in J_1 \right\}. \qquad (37)$$

The index set for the set of pressure nodes on the boundary Γ_1 may be identified as

$$J_B \equiv J \setminus J_1 . \qquad (38)$$

(Note the notational difference between I_B and J_B. The elements of I_B are indices of nodes on Γ_0 and the elements of J_B are indices of nodes on Γ_1.) Finally, the index set for the basis $\{H_i(x)\}$ is defined

$$K = \left\{ 0,1,2,\ldots,N_B - 1 \right\}, \qquad (39)$$

where N_B is the number of nodes in the boundary space $B^h(\Gamma_c)$. The following table summarizes the various node index sets.

Set	Decription
I	Indices of all nodes in the velocity grid
I_0	Indices of velocity nodes *not* on the boundary Γ_0; "internal velocity nodes"
I_B	Indices of velocity nodes *on* the boundary Γ_0
J	Indices of all nodes in the pressure grid
J_1	Indices of pressure nodes *not* on the boundary Γ_1; "internal pressure nodes"
J_B	Indices of pressure nodes on the boundary Γ_1
K	Indices of all nodes on the boundary Γ_c

• Table 2: Node index set descriptions

These sets allow the immediate description of all the approximation spaces for the field variables. These function spaces are collected below in Table 3 for convenience. Compare these spaces with those of Table 1.

Space	Description
$V^h(\Omega)$=span $\{F_i(x)\|i\in I\} \subset V(\Omega)$	Approximation space for $\mathbf{U}(t)$
$V_0^h(\Omega)$=span $\{F_i(x)\|i\in I_0\} \subset V_0(\Omega)$	Space of velocity testing functions
$P^h(\Omega)$=span $\{G_i(x)\|i\in J\} \subset P(\Omega)$	Approximation space for p(t)
$P_1^h(\Omega)$=span $\{G_i(x)\|i\in J_1\} \subset P_1(\Omega)$	Space of pressure testing functions
$B^h(\Gamma_c)$ =span $\{H_i(x)\|i\in K\} \subset B(\Gamma_c)$	Approximation and testing space for $\Lambda(t)$

• Table 3: Approximation function spaces for Navier-Stokes problem

3.2.3 Discrete Equations

Now we insert the function expansions of (28), (29), and (30) into the weak formulation (22) to yield a set of ordinary differential equations. Using $\{F_i(x)\times 0, 0\times F_j(x)|i,j\in I_0\}$ as the set of velocity testing functions (for the first line of Eqs. (22)) we get

$$
\begin{aligned}
&[A_{ij}]\dot{\mathbf{u}}^0 + \nu[B_{ij}]\mathbf{u}^0 + \left([C^X_{ijk}]\mathbf{u}\right)\mathbf{u} + \left([C^Y_{ijk}]\mathbf{v}\right)\mathbf{u} - [D^X_{ij}]\mathbf{p}^1 - [E_{ij}]\boldsymbol{\lambda} = \mathbf{f}^x + \mathbf{W}^X(\mathbf{u}^B,\mathbf{v}^B)\\
&[A_{ij}]\dot{\mathbf{v}}^0 + \nu[B_{ij}]\mathbf{v}^0 + \left([C^X_{ijk}]\mathbf{u}\right)\mathbf{v} + \left([C^Y_{ijk}]\mathbf{v}\right)\mathbf{v} - [D^Y_{ij}]\mathbf{p}^1 - [E_{ij}]\boldsymbol{\mu} = \mathbf{f}^y + \mathbf{W}^Y(\mathbf{u}^B,\mathbf{v}^B)\\
&[D^X_{ij}]^{\mathrm{T}}\mathbf{u}^0 + [D^Y_{ij}]^{\mathrm{T}}\mathbf{v}^0 = 0\\
&[E_{ij}]^{\mathrm{T}}\mathbf{u}^0 = \mathbf{c}\\
&[E_{ij}]^{\mathrm{T}}\mathbf{v}^0 = \mathbf{d}\\
&\mathbf{u}^B = \mathbf{u}^x_{\inf}\\
&\mathbf{v}^B = \mathbf{u}^y_{\inf}\;,
\end{aligned}
\tag{40}
$$

where we have employed the following definitions for the discrete dependent variables:

$$
\begin{array}{lllll}
\mathbf{u} \equiv (u_i\,|i\in I_0\cup I_B) & \mathbf{u}^0 \equiv (u_i\,|i\in I_0) & \mathbf{u}^B \equiv (u_i\,|i\in I_B) & \mathbf{p}^1 \equiv (p_j\,|j\in J_1) & \boldsymbol{\lambda} \equiv (\lambda_k\,|k\in K)\\
\mathbf{v} \equiv (v_i\,|i\in I_0\cup I_B) & \mathbf{v}^0 \equiv (v_i\,|i\in I_0) & \mathbf{v}^B \equiv (v_i\,|i\in I_B) & \mathbf{p}^B \equiv \{p_j|j\in J_B\} & \boldsymbol{\mu} \equiv (\mu_k\,|k\in K)
\end{array}
\tag{41}
$$

The forcing vectors are defined as

$$
\begin{array}{lll}
\mathbf{u}_{\inf} \equiv \left(\mathbf{U}_{\inf}(x_i)\cdot\mathbf{a}_x\,|i\in I_B\right) & \mathbf{f}^x \equiv \left(\int_\Omega f^x F_i\, d\Omega|i\in I_0\right) & \mathbf{c} \equiv \left(\int_{\Gamma_c} c H_i\, d\Gamma\,|i\in K\right)\\
\mathbf{v}_{\inf} \equiv \left(\mathbf{U}_{\inf}(x_i)\cdot\mathbf{a}_y\,|i\in I_B\right) & \mathbf{f}^y \equiv \left(\int_\Omega f^y F_i\, d\Omega|i\in I_0\right) & \mathbf{d} \equiv \left(\int_{\Gamma_c} d H_i\, d\Gamma\,|i\in K\right)
\end{array}
\tag{42}
$$

Note that we have used a simple pointwise approximation scheme for the boundary values on Γ_0. The node values for $\mathbf{U}(x_i)=[u(x_i),v(x_i)]$ are specified directly by the equations $\mathbf{u}^B=\mathbf{u}_{\inf}$ and $\mathbf{v}^B=\mathbf{v}_{\inf}$ rather than using an averaging (integral) approach. (Actually, this is equivalent to using Dirac delta functions as the testing functions.) This makes the system simpler and does not introduce any additional error (this error has the same order as that already incurred by the shape function interpolation scheme).

Note that the advection components in (40) may be decomposed into the internal node and boundary node components as well. The result is given by

$$
\begin{aligned}
\left([C^X_{ijk}]\mathbf{u}\right)\mathbf{u} + \left([C^Y_{ijk}]\mathbf{v}\right)\mathbf{u} = {}& \left([C^X_{ijk}]\mathbf{u}^0\right)\mathbf{u}^0 + \left([C^X_{ijk}]\mathbf{u}^0\right)\mathbf{u}^B + \left([C^X_{ijk}]\mathbf{u}^B\right)\mathbf{u}^0 + \left([C^X_{ijk}]\mathbf{u}^B\right)\mathbf{u}^B\\
&+ \left([C^Y_{ijk}]\mathbf{v}^0\right)\mathbf{u}^0 + \left([C^Y_{ijk}]\mathbf{v}^0\right)\mathbf{u}^B + \left([C^Y_{ijk}]\mathbf{v}^B\right)\mathbf{u}^0 + \left([C^Y_{ijk}]\mathbf{v}^B\right)\mathbf{u}^B\\
\left([C^X_{ijk}]\mathbf{u}\right)\mathbf{v} + \left([C^Y_{ijk}]\mathbf{v}\right)\mathbf{v} = {}& \left([C^X_{ijk}]\mathbf{u}^0\right)\mathbf{v}^0 + \left([C^X_{ijk}]\mathbf{u}^0\right)\mathbf{v}^B + \left([C^X_{ijk}]\mathbf{u}^B\right)\mathbf{v}^0 + \left([C^X_{ijk}]\mathbf{u}^B\right)\mathbf{v}^B\\
&+ \left([C^Y_{ijk}]\mathbf{v}^0\right)\mathbf{v}^0 + \left([C^Y_{ijk}]\mathbf{v}^0\right)\mathbf{v}^B + \left([C^Y_{ijk}]\mathbf{v}^B\right)\mathbf{v}^0 + \left([C^Y_{ijk}]\mathbf{v}^B\right)\mathbf{v}^B
\end{aligned}
$$

The matrices and tensor elements in (40) are given by the values

$$
\begin{aligned}
&A_{ij} = \int_\Omega F_i F_j \, d\Omega && \text{for } i,j \in I && C_{ijk}^X = \int_\Omega F_i \frac{\partial F_j}{\partial x} F_k \, d\Omega && \text{for } i,j \in I \\
&B_{ij} = \int_\Omega \left[\frac{\partial F_i}{\partial x}\frac{\partial F_j}{\partial x} + \frac{\partial F_i}{\partial y}\frac{\partial F_j}{\partial y} \right] d\Omega && \text{for } i,j \in I && C_{ijk}^X = \int_\Omega F_i \frac{\partial F_j}{\partial y} F_k \, d\Omega && \text{for } i,j \in I \\
&D_{ij}^X = \int_\Omega \frac{\partial F_i}{\partial x} G_j \, d\Omega && \text{for } i \in I_0, j \in J && && \\
& && && E_{ij} = \int_{\Gamma_c} F_i H_j \, d\Gamma && \text{for } i \in I, j \in K \\
&D_{ij}^Y = \int_\Omega \frac{\partial F_i}{\partial y} G_j \, d\Omega && \text{for } i \in I_0, j \in J && &&
\end{aligned}
\tag{43}
$$

The functions $\mathbf{W}^X$ and $\mathbf{W}^Y$ in the LHS of (40) represent the contributions due solely to the boundary nodes. The values of these expressions are known functions of $\mathbf{u}^B$ and $\mathbf{v}^B$. They are defined by

$$
\begin{aligned}
&\mathbf{W}^X(\mathbf{u}^B,\mathbf{v}^B) \equiv -[A_{ij}]\dot{\mathbf{u}}^B - [B_{ij}]\mathbf{u}^B - [D_{ij}^X]\mathbf{p}^B - \left\{ \left([C_{ijk}^X]\mathbf{u}^B\right)\mathbf{u}^B + \left([C_{ijk}^Y]\mathbf{v}^B\right)\mathbf{u}^B \right\} \quad \text{where } i \in I_0, j \in I_B \\
&\mathbf{W}^Y(\mathbf{u}^B,\mathbf{v}^B) \equiv -[A_{ij}]\dot{\mathbf{v}}^B - [B_{ij}]\mathbf{v}^B - [D_{ij}^Y]\mathbf{p}^B - \left\{ \left([C_{ijk}^X]\mathbf{u}^B\right)\mathbf{v}^B + \left([C_{ijk}^Y]\mathbf{v}^B\right)\mathbf{v}^B \right\} \quad \text{where } i \in I_0, j \in I_B
\end{aligned}
\tag{44}
$$

where the terms in braces are only added if the advection terms are decomposed into the boundary node and internal node contributions. The conditions at the end of the definitions are meant to indicate that the matrix and tensor elements in the above equations have the same definitions as those in (43) only that the indices belong in the index set I_B rather than I_0. In essence, we are just separating a larger system built from I into the unknown components I_0 and the known components I_B.

3.3 Numerical Integration: Operator-Splitting Methods

Now that we have extracted a discrete approximation to the continuous system we may use established mathematical techniques to solve the system. We employ an operator-splitting technique for the time integration that decomposes the original system into linear and nonlinear subsystems. In this manner we may associate the (difficult) incompressibility condition with the linear system and leave the nonlinear system free (i.e., unconstrained). A conjugate-gradient algorithm solves the resulting linear Stokes problem and a least-squares algorithm may be used to solve the nonlinear advection problem. Here the general properties of using operator-splitting methods for numerical integration are covered. In the following section we apply the techniques to the discretized Navier-Stokes system.

3.3.1 Operator-Splitting Methods

Operator splitting methods are a family of techniques for numerically integrating general operator equations. They are based on the notion of decomposing an (possibly nonlinear) operator into separate components which are individually simpler than the original. To demonstrate this numerical integration technique, we shall use the following system as a model. Consider the operator equation on the Hilbert space H

$$
\begin{aligned}
\dot{\phi} + \mathrm{A}\phi &= 0 \\
\phi(0) &= \phi_o
\end{aligned}
\tag{45}
$$

where $A{:}H \to H$ is a (possibly nonlinear) operator on H, $\varphi(t) \in H$, and the over dot indicates time differentiation. Now suppose A may be nontrivially decomposed as

$$A = A_1 + A_2 . \qquad (46)$$

The above system is to be integrated using operator splitting. A straightforward technique is the Peaceman-Rachford scheme [5] which we shall describe. First, however, we must introduce the discrete time and its notation.

A finite-length time step $\Delta t > 0$ is chosen. The dependent variable φ is then indexed for the discrete times $n\Delta t$ according to the following:

$$\phi_n \equiv \phi(n\,\Delta t) . \qquad (47)$$

NOTE:

Another potential technique for solving such problems is the use of semigroup methods. This idea is based on the generalized solution of the operator equation $\dot{\phi} = \mathbf{A}\phi$ which is represented abstractly as $\phi(t) = e^{\mathbf{A}t}\phi(0)$. These techniques have been well studied for linear systems [10],[11].

3.3.2 The Peaceman-Rachford Scheme

With the Peaceman-Rachford scheme we divide the time interval $[n\Delta t,(n+1)\Delta t]$ into two subintervals $[n\Delta t,(n+1/2)\Delta t]$ and $[(n+1/2)\Delta t,(n+1)\Delta t]$. Assuming that φ_n is known, we approximate the value of $\varphi_{n+1/2}$ using a backward Euler step with respect to A_1 and a forward Euler step with respect to A_2. Then the value of φ_{n+1} is approximated from $\varphi_{n+1/2}$ using the same process only with the roles of A_1 and A_2 reversed. The result is that

$$\begin{aligned} &\frac{\phi_{n+1/2} - \phi_n}{\Delta t / 2} + A_1(\phi_{n+1/2}) + A_2(\phi_n) = 0, \\ &\frac{\phi_{n+1} - \phi_{n+1/2}}{\Delta t / 2} + A_1(\phi_{n+1/2}) + A_2(\phi_{n+1}) = 0, \end{aligned} \qquad (48)$$

must be solved successively for $\varphi_{n+1/2}$ and φ_{n+1}. Note that the above is an implicit scheme. However, the idea here is to choose A_1 and A_2 so that the nonlinear and linear components are separated making the solution of the above easier.

We examine the stability and accuracy of the Peaceman-Rachford scheme by considering a simple example. Let $H = \mathbf{R}^N$, $\varphi \in \mathbf{R}^N$, and $A = [A_{ij}]$ be a positive-definite, symmetric $N \times N$ matrix. The solution of Eq. (45) is then given by

$$\phi(t) = e^{-[A_{ij}]t}\phi_0 \qquad (49)$$

which may be decomposed into modal equations using the eigenvalues, $\{\lambda_i\}$, and eigenvectors, $\{e_i\}$, of $[A_{ij}]$. The solution in terms of the component modes is given by

$$\phi(t) = \sum_{i=1}^{N} \phi^i(t) \qquad \text{where} \qquad \phi^i(t) = e^{-\lambda_i t}\phi_0^i \;\; i = 1, \ldots, N. \qquad (50)$$

The quantity $\varphi^i(t)$ is the modal component of $\varphi(t)$ for the eigenvalue λ_i (i.e., the projection of $\varphi(t)$ onto e_i) and φ_0^i is the component of φ_0 in the e_i direction. By linearity we are able to track the modes separately. To apply the Peaceman-Rachford method we impose the following decomposition of the matrix $[A_{ij}]$:

$$[A_{ij}] = \alpha[A_{ij}] + \beta[A_{ij}] \qquad (51)$$

where $\alpha,\beta \in (0,1)$ with $\alpha+\beta=1$. Solving Eqs. (48) for φ_{n+1} with $A_1=\alpha[A_{ij}]$ and $A_2=\beta[A_{ij}]$ yields

$$\phi_{n+1} = \left([\delta_{ij}] + \frac{\Delta t}{2}\beta[A_{ij}]\right)^{-1}\left([\delta_{ij}] - \frac{\Delta t}{2}\alpha[A_{ij}]\right)\left([\delta_{ij}] + \frac{\Delta t}{2}\alpha[A_{ij}]\right)^{-1}\left([\delta_{ij}] - \frac{\Delta t}{2}\beta[A_{ij}]\right)\phi_n \quad (52)$$

(δ_{ij} is the Kronecker delta function) so that the discrete approximation to Eq. (49) is given by

$$\phi_n = \left([\delta_{ij}] + \frac{\Delta t}{2}\beta[A_{ij}]\right)^{-n}\left([\delta_{ij}] - \frac{\Delta t}{2}\alpha[A_{ij}]\right)^{n}\left([\delta_{ij}] + \frac{\Delta t}{2}\alpha[A_{ij}]\right)^{-n}\left([\delta_{ij}] - \frac{\Delta t}{2}\beta[A_{ij}]\right)^{n}\phi_0 \,. \quad (53)$$

Doing a modal expansion for this discrete solution yields the following solution in terms of the modal components φ_n^i of φ_n:

$$\phi_n^i = \left(\frac{1-\frac{\Delta t}{2}\alpha\lambda_i}{1+\frac{\Delta t}{2}\alpha\lambda_i}\right)^n\left(\frac{1-\frac{\Delta t}{2}\beta\lambda_i}{1+\frac{\Delta t}{2}\beta\lambda_i}\right)^n \phi_0^i \quad \text{for } i = 1,\ldots,N. \qquad (54)$$

Since $|(1-x)/(1+x)|<1$ for all $x>0$, the above equation shows that $|\varphi_n^i|\leq|\varphi_0^i|$ for all discrete times $n\geq 0$. This fact implies the stability of the Peaceman-Rachford method, at least for linear systems. The above equation also implies $\lim_{n\to\infty}\varphi_n^i=0$ for all i, which we expect from Eq. (49) and the positive definiteness of $[A_{ij}]$. To determine the accuracy, set $\mu=\Delta t\lambda_i$ and introduce the rational function $R(\mu)$ defined by

$$R(\mu) \equiv \left(\frac{1-\frac{\alpha}{2}\mu}{1+\frac{\alpha}{2}\mu}\right)\left(\frac{1-\frac{\beta}{2}\mu}{1+\frac{\beta}{2}\mu}\right). \qquad (55)$$

Talyor expanding $R(\mu)$ about the point $\mu=0$ yields

$$R(\mu) = 1-\mu+\frac{1}{2}\mu^2 - \frac{\alpha^2+\alpha\beta+\beta^2}{4}\mu^3 + O(\mu^4) \qquad (56)$$

which when compared to the Taylor expansion of $e^{-\mu}$ about $\mu=0$,

$$e^{-\mu} = 1-\mu+\frac{1}{2}\mu^2 - \frac{1}{6}\mu^3 + O(\mu^4), \qquad (57)$$

shows that we have at least second-order accuracy for any values of α and β. The best values of α and β conforming to the constraint $\alpha+\beta=1$ are $\alpha=\beta=1/2$. This gives us "almost" third order accuracy since $(\alpha^2+\alpha\beta+\beta^2)/4=3/16=1/6+1/48$.

The definition of $R(\mu)$ also points out the main disadvantage of the Peaceman-Rachford method, namely, it converges slowly for stiff systems. For large values of μ, and consequently for large values of $\Delta t\lambda_i$, the value of $R(\mu)$ approaches unity. This implies that φ_n^i converges slowly to zero as $n\rightarrow\infty$. From Eq. (50) we know that the true solution $\varphi^i(t)$ will converge rapidly to zero as $t\rightarrow\infty$. Therefore, the Peaceman-Rachford method is not able to capture fast transients unless Δt is taken to be extremely small. This condition is also true of the Crank-Nicolson integration method.

3.3.3 The θ-Scheme

The θ-scheme is a generalization of the Peaceman-Rachford scheme and was introduced by Glowinski in the mid-1980s [12]. In this method the time interval $[n\Delta t,(n+1)\Delta t]$ is broken into three subintervals. The result is an unconditionally stable integration scheme which is able to capture the behavior of stiff systems much more accurately when using larger time-steps Δt.

Again we use Eq. (45) as our model system. The θ-scheme begins by picking a parameter $\theta\in(0,1/2)$ and a time step $\Delta t>0$. Assuming that φ_n is known, we then solve the following system, consecutively, for $\varphi_{n+\theta}$, $\varphi_{n+1-\theta}$, and φ_{n+1}:

$$\begin{aligned}
&\frac{\phi_{n+\theta}-\phi_n}{\theta\Delta t}+A_1(\phi_{n+\theta})+A_2(\phi_n)=0,\\
&\frac{\phi_{n+1-\theta}-\phi_{n+\theta}}{(1-2\theta)\Delta t}+A_1(\phi_{n+\theta})+A_2(\phi_{n+1-\theta})=0, \qquad (58)\\
&\frac{\phi_{n+1}-\phi_{n+1-\theta}}{(1-2\theta)\Delta t}+A_1(\phi_{n+1})+A_2(\phi_{n+1-\theta})=0.
\end{aligned}$$

To check the performance of this algorithm with respect to the Peaceman-Rachford method again consider the simple situation of $H=\mathbf{R}^N$, $\varphi\in\mathbf{R}^N$, and $A=[A_{ij}]$ a positive-definite, symmetric $N\times N$ matrix. We proceed as before, solving Eqs. (58) for φ_{n+1} with $A_1=\alpha[A_{ij}]$ and $A_2=\beta[A_{ij}]$ where $\alpha,\beta\in(0,1)$, $\alpha+\beta=1$. Setting $\theta'=1-2\theta$, the solution is

$$\phi_{n+1}=\left([\delta_{ij}]+\alpha\theta\Delta t[A_{ij}]\right)^{-2}\left([\delta_{ij}]-\beta\theta\Delta t[A_{ij}]\right)^{2}\left([\delta_{ij}]+\beta\theta'\Delta t[A_{ij}]\right)^{-1}\left([\delta_{ij}]-\alpha\theta'\Delta t[A_{ij}]\right)\phi_n \qquad (59)$$

and the corresponding discrete modal decomposition is

$$\phi_n^i=\frac{(1-\beta\theta\Delta t\lambda_i)^{2n}}{(1+\alpha\theta\Delta t\lambda_i)^{2n}}\frac{(1-\alpha\theta'\Delta t\lambda_i)^n}{(1+\beta\theta'\Delta t\lambda_i)^n}\phi_0^i \quad \text{for } i=1,\ldots,N. \qquad (60)$$

To explore the stability for stiff systems define the rational function $R'(\mu)$

$$R'(\mu)\equiv\frac{(1-\beta\theta\mu)^2}{(1+\alpha\theta\mu)^2}\frac{(1-\alpha\theta'\mu)}{(1+\beta\theta'\mu)}. \qquad (61)$$

Since

$$|R'(\mu)|\xrightarrow[\mu\to\infty]{}\frac{\beta}{\alpha}, \qquad (62)$$

requiring α>β should insure better convergence than the Peaceman-Rachford scheme when integrating stiff systems. In fact, the larger the value of α compared to β the better the convergence properties. However, we should also consider the accuracy of the θ-scheme. To do this Taylor expand $R(\mu)$ about $\mu=0$ to obtain

$$R'(\mu) = 1 - \mu + \frac{1}{2}\left[1 + (\beta^2 - \alpha^2)(2\theta^2 - 4\theta + 1)\right]\mu^2 + O(\mu^3). \qquad (63)$$

From here we see that if α=β or

$$\theta = 1 - \frac{1}{\sqrt{2}} \approx 0.292893186 \qquad (64)$$

then the method is second-order accurate. If, however, neither of these conditions hold then the method is only first-order accurate. Thus, a reasonable tradeoff between stability accuracy seems to be the choice given by Eq. (64). Also, it is convenient to pick α and β so that the same matrix is obtained for each integration step in (58). This means that $\alpha\theta=\beta(1-2\theta)$ from which we may determine (requiring α>β)

$$\alpha = \frac{1-2\theta}{1-\theta} \quad \text{and} \quad \beta = \frac{\theta}{1-\theta}. \qquad (65)$$

For the value of θ given by Eq. (64) the values of α and β are then

$$\alpha = 2 - \sqrt{2} \approx 0.5857864380 \quad \text{and} \quad \beta = \sqrt{2} - 1 \approx 0.4142135620. \qquad (66)$$

This gives $\beta/\alpha=1/\sqrt{2}\approx 0.707107$.

3.4 Numerical Integration of the Navier-Stokes System

The application of the θ-scheme for time integrating the discrete Navier-Stokes system is outlined here. For a more complete description see reference [13]. The discrete-time values for the dependent variables are, as before, denoted with a subscript. For example, the *x*-directed flow is given by

$$\mathbf{u}_n^0 \equiv \mathbf{u}^0(n\,\Delta t). \qquad (67)$$

The other dependent variables are time indexed similarly. We assume that the values of the dependent variables are given for $n=0$, that is, we have a divergence-free flow from which to start. Again introducing the numerical parameters $\theta\in(0,1/2)$ and $\alpha,\beta\in(0,1)$ with $\alpha+\beta=1$, we must solve the following three systems:

$$[A_{ij}]\frac{\mathbf{u}^0_{n+\theta}-\mathbf{u}^0_n}{\theta\,\Delta t}+\alpha\nu[B_{ij}]\mathbf{u}^0_{n+\theta}-[D^X_{ij}]\mathbf{p}^1_{n+\theta}-[E_{ij}]\boldsymbol{\lambda}_{n+\theta}=$$

$$\mathbf{f}^{\mathbf{x}}_{n+\theta}+\mathbf{W}^X(\mathbf{u}^B_{n+\theta},\mathbf{v}^B_{n+\theta})-\beta\nu[B_{ij}]\mathbf{u}^0_n-\left([C^X_{ijk}]\mathbf{u}_n\right)\mathbf{u}_n-\left([C^Y_{ijk}]\mathbf{v}_n\right)\mathbf{u}_n$$

$$[A_{ij}]\frac{\mathbf{v}^0_{n+\theta}-\mathbf{v}^0_n}{\theta\,\Delta t}+\alpha\nu[B_{ij}]\mathbf{v}^0_{n+\theta}-[D^Y_{ij}]\mathbf{p}^1_{n+\theta}-[E_{ij}]\boldsymbol{\mu}_{n+\theta}=$$

$$\mathbf{f}^{\mathbf{x}}_{n+\theta}+\mathbf{W}^Y(\mathbf{u}^B_{n+\theta},\mathbf{v}^B_{n+\theta})-\beta\nu[B_{ij}]\mathbf{v}^0_{n+\theta}-\left([C^X_{ijk}]\mathbf{u}_n\right)\mathbf{v}_n-\left([C^Y_{ijk}]\mathbf{v}_n\right)\mathbf{v}_n$$

$$[D^X_{ij}]^{\mathrm{T}}\mathbf{u}^0_{n+\theta}+[D^Y_{ij}]^{\mathrm{T}}\mathbf{v}^0_{n+\theta}=0 \tag{68}$$

$$[E_{ij}]^{\mathrm{T}}\mathbf{u}^0_{n+\theta}=\mathbf{c}_{n+\theta}$$

$$[E_{ij}]^{\mathrm{T}}\mathbf{v}^0_{n+\theta}=\mathbf{d}_{n+\theta}$$

$$\mathbf{u}^{\mathrm{B}}_{n+\theta}=\mathbf{u}_{\mathrm{inf}}$$

$$\mathbf{v}^{\mathrm{B}}_{n+\theta}=\mathbf{v}_{\mathrm{inf}}$$

for $\mathbf{u}_{n+\theta}{}^0$, $\mathbf{v}_{n+\theta}{}^0$, $\mathbf{p}_{n+\theta}$, $\lambda_{n+\theta}$, and $\mu_{n+\theta}$; next solve

$$[A_{ij}]\frac{\mathbf{u}^0_{n+1-\theta}-\mathbf{u}^0_{n+\theta}}{(1-2\theta)\,\Delta t}+\beta\nu[B_{ij}]\mathbf{u}^0_{n+1-\theta}+\left([C^X_{ijk}]\mathbf{u}_{n+1-\theta}\right)\mathbf{u}_{n+1-\theta}+\left([C^Y_{ijk}]\mathbf{v}_{n+1-\theta}\right)\mathbf{u}_{n+1-\theta}=$$

$$\mathbf{f}^{\mathbf{x}}_{n+1-\theta}+\mathbf{W}^X(\mathbf{u}^B_{n+1-\theta},\mathbf{v}^B_{n+1-\theta})-\alpha\nu[B_{ij}]\mathbf{u}^0_{n+\theta}-[D^X_{ij}]\mathbf{p}^1_{n+\theta}-[E_{ij}]\boldsymbol{\lambda}_{n+\theta}$$

$$[A_{ij}]\frac{\mathbf{v}^0_{n+1-\theta}-\mathbf{v}^0_{n+\theta}}{(1-2\theta)\,\Delta t}+\beta\nu[B_{ij}]\mathbf{v}^0_{n+1-\theta}+\left([C^X_{ijk}]\mathbf{u}_{n+1-\theta}\right)\mathbf{v}_{n+1-\theta}+\left([C^Y_{ijk}]\mathbf{v}_{n+1-\theta}\right)\mathbf{v}_{n+1-\theta}= \tag{69}$$

$$\mathbf{f}^{\mathbf{x}}_{n+1-\theta}+\mathbf{W}^Y(\mathbf{u}^{\mathrm{B}}_{n+1-\theta},\mathbf{v}^{\mathrm{B}}_{n+1-\theta})-\alpha\nu[B_{ij}]\mathbf{v}^0_{n+\theta}-[D^Y_{ij}]\mathbf{p}^1_{n+\theta}-[E_{ij}]\boldsymbol{\mu}_{n+\theta}$$

$$\mathbf{u}^{\mathrm{B}}_{n+1-\theta}=\mathbf{u}_{\mathrm{inf}}$$

$$\mathbf{v}^{\mathrm{B}}_{n+1-\theta}=\mathbf{v}_{\mathrm{inf}}$$

for $\mathbf{u}_{n+1-\theta}{}^0$ and $\mathbf{v}_{n+1-\theta}{}^0$; finally solve

$$[A_{ij}]\frac{\mathbf{u}^0_{n+1}-\mathbf{u}^0_{n+1-\theta}}{\theta\,\Delta t}+\alpha\nu[B_{ij}]\mathbf{u}^0_{n+1}-[D^X_{ij}]\mathbf{p}^1_{n+1}-[E_{ij}]\boldsymbol{\lambda}_{n+1}=$$

$$\mathbf{f}^{\mathbf{x}}_{n+1}+\mathbf{W}^X(\mathbf{u}^B_{n+1},\mathbf{v}^B_{n+1})-\beta\nu[B_{ij}]\mathbf{u}^0_{n+1-\theta}-\left([C^X_{ijk}]\mathbf{u}_{n+1-\theta}\right)\mathbf{u}_{n+1-\theta}-\left([C^Y_{ijk}]\mathbf{v}_{n+1-\theta}\right)\mathbf{u}_{n+1-\theta}$$

$$[A_{ij}]\frac{\mathbf{v}^0_{n+1}-\mathbf{v}^0_{n+1-\theta}}{\theta\,\Delta t}+\alpha\nu[B_{ij}]\mathbf{v}^0_{n+1}-[D^Y_{ij}]\mathbf{p}^1_{n+1}-[E_{ij}]\boldsymbol{\mu}_{n+1}=$$

$$\mathbf{f}^{\mathbf{x}}_{n+1}+\mathbf{W}^Y(\mathbf{u}^{\mathrm{B}}_{n+1},\mathbf{v}^{\mathrm{B}}_{n+1})-\beta\nu[B_{ij}]\mathbf{v}^0_{n+1-\theta}-\left([C^X_{ijk}]\mathbf{u}_{n+1-\theta}\right)\mathbf{v}_{n+1-\theta}-\left([C^Y_{ijk}]\mathbf{v}_{n+1-\theta}\right)\mathbf{v}_{n+1-\theta}$$

$$[D^X_{ij}]^{\mathrm{T}}\mathbf{u}^0_{n+1}+[D^Y_{ij}]^{\mathrm{T}}\mathbf{v}^0_{n+1}=0 \tag{70}$$

$$[E_{ij}]^{\mathrm{T}}\mathbf{u}^0_{n+1}=\mathbf{c}_{n+1}$$

$$[E_{ij}]^{\mathrm{T}}\mathbf{v}^0_{n+1}=\mathbf{d}_{n+1}$$

$$\mathbf{u}^{\mathrm{B}}_{n+1}=\mathbf{u}_{\mathrm{inf}}$$

$$\mathbf{v}^{\mathrm{B}}_{n+1}=\mathbf{v}_{\mathrm{inf}}$$

for $\mathbf{u}_{n+1}{}^0$, $\mathbf{v}_{n+1}{}^0$, $\mathbf{p}_{n+1}$, λ_{n+1}, and μ_{n+1}. For the particular choice of integration parameters given by Eqs. (65) and (66) (i.e., $\theta=1-1/\sqrt{2}$, $\alpha=2-\sqrt{2}$, and $\beta=\sqrt{2}-1$), the θ-scheme for the Navier-Stokes system seems unconditionally stable [14].

3.4.1 Solution of the Stokes Problem

For each time step, two Stokes-type systems and one advection-type problem must be solved. The Stokes systems include the incompressibility condition which, in the discrete system, is an algebraic constraint. Here we develop an iterative method for solving the Stokes' systems; it is based on a one-shot, conjugate-gradient algorithm with preconditioning. The method is driven by the Lagrange multipliers $\mathbf{p}_n$, λ_n, and μ_n [15]. The solution of the advection subproblem is outlined in the following section. We begin by describing the direct solution to the Stokes problem and its relationship to some simple iterative schemes.

The discrete Stokes system may be written in the form

$$
\begin{aligned}
&\delta[A_{ij}]\mathbf{u} + \nu[B_{ij}]\mathbf{u} = \mathbf{f}^x + [D_{ij}^X]\mathbf{p} + [E_{ij}]\lambda \\
&\delta[A_{ij}]\mathbf{v} + \nu[B_{ij}]\mathbf{v} = \mathbf{f}^y + [D_{ij}^Y]\mathbf{p} + [E_{ij}]\mu \\
&[D_{ij}^X]^{\mathrm{T}}\mathbf{u} + [D_{ij}^Y]^{\mathrm{T}}\mathbf{v} = 0 \qquad (71) \\
&[E_{ij}]^{\mathrm{T}}\mathbf{u} = \mathbf{c} \\
&[E_{ij}]^{\mathrm{T}}\mathbf{v} = \mathbf{d}
\end{aligned}
$$

where δ is the reciprocal of the time step and where we have generalized $\mathbf{f}^x$ and $\mathbf{f}^y$ to include the entire right-hand sides of Eqs. (68) and (70). This is the form on which we concentrate.

In Phase I we used direct methods to solve the Stokes problem. Since we were considering small systems for the purpose of development, this presented little problem. However, when considering real-world situations the system size will typically make direct solution methods so computationally intense as to place a bottleneck on real-time computation. For large systems it is more practical to use iterative solution techniques. Such techniques are usually based on the principle of minimizing a functional of the dependent variables. The process is designed so that the functional's minimum corresponds to the solution of the original system. Therefore, in this subsection we present the direct solution, a general iterative scheme, and a preconditioned conjugate gradient scheme developed by Glowinski, et al [14]. With preconditioning, he reports an effective decoupling between system size and number of iterations until convergence.

3.4.1.1 The Direct Solution

The direct solution to (71) may be found by forming an augmented linear system with dependent variable $(\mathbf{u},\mathbf{v},\mathbf{p},\lambda,\mu)^{\mathrm{T}}$. First, to simplify matters we define the matrix $[T_{ij}]$ according to

$$[T_{ij}] \equiv \delta[A_{ij}] + \nu[B_{ij}]. \qquad (72)$$

Note that, in general, $[T_{ij}]$ is invertible. Now we collect all the equations in (50) into a single matrix-vector equation.

$$\begin{pmatrix} [T_{ij}] & \mathbf{0} & -[D_{ij}^X] & -[E_{ij}] & \mathbf{0} \\ \mathbf{0} & [T_{ij}] & -[D_{ij}^Y] & \mathbf{0} & -[E_{ij}] \\ -[D_{ij}^X]^T & -[D_{ij}^Y]^T & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ -[E_{ij}]^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -[E_{ij}]^T & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{p} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \mathbf{f}^{\mathbf{x}} \\ \mathbf{f}^{\mathbf{y}} \\ \mathbf{0} \\ -\mathbf{c} \\ -\mathbf{d} \end{pmatrix}. \tag{73}$$

Obviously, the direction solution of the above would not be the most efficient way to solve the problem on a computer. The matrix in (73) is very sparse; indeed, the submatrices themselves are sparse. Therefore, when solving the system directly it is wise to exploit the structure of the above system. However, from immediate inspection of Eq. (73) we do get some clues as to the behavior of our system. For one, it is symmetric, or self-adjoint. Therefore, the eigenvalues will be real and distinct, assuming the system is of full rank. We know from the governing equations that the pressure p is only defined to within an arbitrary constant, so it might be the case that the above is not of full rank. However, we show below that the matrices $[D_{ij}^X]$, $[D_{ij}^Y]$, and $[E_{ij}]$ are of full rank and therefore (73) has a unique solution.

Since the matrix $[T_{ij}]$ is invertible, we can solve $\mathbf{u}$ and $\mathbf{v}$ in terms of the remaining dependent variables using the first two rows in (73). The result is

$$\begin{aligned} \mathbf{u} &= [T_{ij}]^{-1}\mathbf{f}^{\mathbf{x}} + [T_{ij}]^{-1}[D_{ij}^X]\mathbf{p} + [T_{ij}]^{-1}[E_{ij}]\boldsymbol{\lambda}, \\ \mathbf{v} &= [T_{ij}]^{-1}\mathbf{f}^{\mathbf{y}} + [T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p} + [T_{ij}]^{-1}[E_{ij}]\boldsymbol{\mu}. \end{aligned} \tag{74}$$

By substituting these values into the forth and fifth row equations we get the equations for the control constraints

$$\begin{aligned} [E_{ij}]^T[T_{ij}]^{-1}\mathbf{f}^{\mathbf{x}} + [E_{ij}]^T[T_{ij}]^{-1}[D_{ij}^X]\mathbf{p} + [E_{ij}]^T[T_{ij}]^{-1}[E_{ij}]\boldsymbol{\lambda} &= \mathbf{c}, \\ [E_{ij}]^T[T_{ij}]^{-1}\mathbf{f}^{\mathbf{y}} + [E_{ij}]^T[T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p} + [E_{ij}]^T[T_{ij}]^{-1}[E_{ij}]\boldsymbol{\mu} &= \mathbf{d}. \end{aligned} \tag{75}$$

The matrix $[E_{ij}]$ is of full rank since for every i we can find a j such that $\int H_i F_j d\Gamma > 0$ (i.e., the boundary Γ_c intersects the domain Ω somewhere). Therefore the matrix $[E_{ij}]^T[E_{ij}]$ is square, of full rank, and, thus, invertible. Likewise since $[T_{ij}]^{-1}$ is of full rank, the matrix $[E_{ij}]^T[T_{ij}]^{-1}[E_{ij}]$ is also invertible. Using a similar arguments we can make the same conclusions about the matrices $[D_{ij}^X]^T[T_{ij}]^{-1}[D_{ij}^X]$ and $[D_{ij}^Y]^T[T_{ij}]^{-1}[D_{ij}^Y]$. This motivates the convenient definitions

$$\begin{aligned} [\Xi_{ij}] &\equiv [E_{ij}]^T[T_{ij}]^{-1}[E_{ij}], \\ [\Delta_{ij}^X] &\equiv [D_{ij}^X]^T[T_{ij}]^{-1}[D_{ij}^X], \\ [\Delta_{ij}^Y] &\equiv [D_{ij}^Y]^T[T_{ij}]^{-1}[D_{ij}^Y] \\ [\Delta_{ij}] &\equiv [\Delta_{ij}^X] + [\Delta_{ij}^Y]. \end{aligned} \tag{76}$$

Using the fact that all the above matrices are invertible, we may begin solving for the dependent variables in (73). From Eq. (75) we may solve for λ and μ in terms of $\mathbf{p}$

$$\lambda = -[\Xi_{ij}]^{-1}[E_{ij}]^T[T_{ij}]^{-1}[D_{ij}^X]\mathbf{p} + [\Xi_{ij}]^{-1}\mathbf{c} - [\Xi_{ij}]^{-1}[E_{ij}]^T[T_{ij}]^{-1}\mathbf{f}^x,$$
$$\mu = -[\Xi_{ij}]^{-1}[E_{ij}]^T[T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p} + [\Xi_{ij}]^{-1}\mathbf{d} - [\Xi_{ij}]^{-1}[E_{ij}]^T[T_{ij}]^{-1}\mathbf{f}^y. \qquad (77)$$

Finally, the pressure **p** is found from the third row of Eq. (73) and the above relations. We have

$$\mathbf{p} = -\left([\Delta_{ij}^X] + [\Delta_{ij}^Y]\right)^{-1}\left([D_{ij}^X]^T[T_{ij}]^{-1}[E_{ij}]\lambda + [D_{ij}^Y]^T[T_{ij}]^{-1}[E_{ij}]\mu + [D_{ij}^X]^T[T_{ij}]^{-1}\mathbf{f}^x + [D_{ij}^Y]^T[T_{ij}]^{-1}\mathbf{f}^y\right). \qquad (78)$$

Equations (74), (77), and (78) may be back substituted to find independent expressions for all the dependent variables.

3.4.1.2 Overview of Iterative Algorithms for Linear Systems

Typically, iterative schemes for solving linear systems can be formulated in terms of the minimization of a functional. For example, consider the simple matrix-vector system

$$[A_{ij}]\mathbf{x} = \mathbf{y} \qquad (79)$$

Where **x** is the unknown vector and **y** is given. We shall assume that $[A_{ij}]$ is symmetric and positive definite. The solution is, of course, $\mathbf{x}=[A_{ij}]^{-1}\mathbf{y}$. However, rather than computing this directly we choose to minimize the functional

$$f(\mathbf{x}) \equiv \frac{1}{2}\mathbf{x}^T[A_{ij}]\mathbf{x} - \mathbf{x}^T\mathbf{y} + \mathbf{c}, \qquad (80)$$

where **c** is some constant vector. Notice that since $[A_{ij}]$ is positive definite the solution to the minimization problem is $\mathbf{x}=[A_{ij}]^{-1}\mathbf{y}$, which is found by direct differentiation. The idea here is to iteratively update the current solution vector $\mathbf{x}_m$ according to some (intelligent) process guided by the minimization of $f(\mathbf{x})$. More specifically, we update $\mathbf{x}_m$ according to the formula

$$\mathbf{x}_{m+1} = \mathbf{x}_m - \alpha_m\mathbf{d}_m \qquad (81)$$

where $\mathbf{d}_m$ is the current search direction and $\alpha_m>0$ is the search length in the direction $\mathbf{d}_m$. The art of the algorithm is in the choice of the search direction $\mathbf{d}_m$. The search direction may be found by direct scalar minimization of Eq. (80). Specifically, define the scalar function $\varphi(\alpha_m)$ according to

$$\begin{aligned}\phi(\alpha_m) &\equiv f(\mathbf{x}_m - \alpha_m\mathbf{d}_m) \\ &= \frac{1}{2}\alpha_m^2\mathbf{d}_m^T[A_{ij}]\mathbf{d}_m - \alpha_m\mathbf{d}_m^T[A_{ij}]\mathbf{x}_m + \alpha_m\mathbf{d}_m^T\mathbf{y} + \frac{1}{2}\mathbf{x}_m^T[A_{ij}]\mathbf{x}_m - \mathbf{x}_m^T\mathbf{y} + \mathbf{c}\end{aligned} \qquad (82)$$

where we have used the fact that $[A_{ij}]$ is symmetric (as in the fluids case). By differentiating $\varphi(\alpha_m)$ with respect to α_m and setting the result equal to zero, we find the minimizing value of α_m to be

$$\alpha_m = \frac{\mathbf{d}_m^T\left([A_{ij}]\mathbf{x}_m - \mathbf{y}\right)}{\mathbf{d}_m^T[A_{ij}]\mathbf{d}_m}. \qquad (83)$$

From the above we see that the value of α_m becomes zero once the $\mathbf{x}_m = \boldsymbol{x}$ is reached. This has an interesting interpretation in the inner product space generated by $[A_{ij}]$. Define the inner product $\langle\cdot,\cdot\rangle_A$ and induced norm $|\cdot|_A$ by

$$\langle \mathbf{x},\mathbf{y}\rangle_A \equiv \mathbf{x}^T[A_{ij}]\mathbf{y},$$
$$\text{and} \tag{84}$$
$$|\mathbf{x}|_A \equiv \langle \mathbf{x},\mathbf{x}\rangle_A^{\;1/2} = \left[\mathbf{x}^T[A_{ij}]\mathbf{x}\right]^{1/2}.$$

Since we have assumed that $[A_{ij}]$ is symmetric and positive definite we know that this does indeed form a valid inner product. Now consider the unit vector in the direction of $\mathbf{d}_m$ given by

$$\overline{\mathbf{d}}_m \equiv \frac{\mathbf{d}_m}{|\mathbf{d}_m|_A} = \frac{\mathbf{d}_m}{\left[\mathbf{d}_m^T[A_{ij}]\mathbf{d}_m\right]^{1/2}}. \tag{85}$$

In this context, $\alpha_m\mathbf{d}_m$ may be expressed

$$\alpha_m\mathbf{d}_m = \overline{\mathbf{d}}_m^T\left([A_{ij}]\mathbf{x}_m - \mathbf{y}\right)\overline{\mathbf{d}}_m = \left\langle \overline{\mathbf{d}}_m, \mathbf{x}_m - [A_{ij}]^{-1}\mathbf{y}\right\rangle \overline{\mathbf{d}}_m \tag{86}$$

so that

$$\begin{aligned}\mathbf{x}_{m+1} &= \mathbf{x}_m - \left\langle \overline{\mathbf{d}}_m, \mathbf{x}_m - [A_{ij}]^{-1}\mathbf{y}\right\rangle \overline{\mathbf{d}}_m, \\ &= \mathbf{x}_m + \mathrm{Proj}_{\overline{\mathbf{d}}_m}\left([A_{ij}]^{-1}\mathbf{y} - \mathbf{x}_m\right)\overline{\mathbf{d}}_m.\end{aligned} \tag{87}$$

Thus, we see that at each iteration we are adding in the offset of $\mathbf{x}_m$ (from the solution value $\boldsymbol{x}=[A_{ij}]^{-1}\mathbf{y}$) along the search direction $\mathbf{d}_m$. Consider if the search directions $\{\mathbf{d}_m\}$ could be chosen so that they were orthogonal in the inner product $\langle\cdot,\cdot\rangle_A$. We say that the $\{\mathbf{d}_m\}$ are $[A_{ij}]$-orthogonal or "$[A_{ij}]$-conjugate". Then $\{\mathbf{d}_m\}_{m=1}^{N}$ would form a basis of the inner product space and the algorithm would be guaranteed to converge in N steps, where $N=size(\mathbf{x}_m)$. This tenet is the foundation of the conjugate-gradient algorithms.

3.4.1.3 A Functional Equation for the Stokes System

Now consider the discretized Navier-Stokes system. Notice that if we know $(\mathbf{p},\lambda,\mu)$ then we can find $(\mathbf{u},\mathbf{v})$ by direct substitution from Eqs. (53). This suggests an iterative scheme for solving the above system. We can create a functional, which when minimized, yields the solution to the linear system of Eq. (73). This functional, f, will be a function of the variables $(\mathbf{p},\lambda,\mu)$ only. To exemplify this idea we simplify the situation by removing the solid object in the fluid domain so that $\lambda=\mu=\mathbf{0}$. In this case we find from Eq. (78) that

$$\mathbf{p} = -\left([\Delta_{ij}^X]+[\Delta_{ij}^Y]\right)^{-1}\left([D_{ij}^X]^T[T_{ij}]^{-1}\mathbf{f}^{\mathbf{x}} + [D_{ij}^Y]^T[T_{ij}]^{-1}\mathbf{f}^{\mathbf{y}}\right). \tag{88}$$

However, we wish to find this solution iteratively. So, in the spirit of Eq. (80), we form the functional

$$
f(\mathbf{u},\mathbf{v},\mathbf{p}) = \frac{1}{2}\begin{pmatrix}\mathbf{u}\\ \mathbf{v}\\ \mathbf{p}\end{pmatrix}^T \begin{pmatrix} [T_{ij}] & \mathbf{0} & -[D_{ij}^X] \\ \mathbf{0} & [T_{ij}] & -[D_{ij}^Y] \\ -[D_{ij}^X]^T & -[D_{ij}^Y]^T & \mathbf{0} \end{pmatrix}\begin{pmatrix}\mathbf{u}\\ \mathbf{v}\\ \mathbf{p}\end{pmatrix} - \begin{pmatrix}\mathbf{u}\\ \mathbf{v}\\ \mathbf{p}\end{pmatrix}^T\begin{pmatrix}\mathbf{f}^x\\ \mathbf{f}^y\\ \mathbf{0}\end{pmatrix},
$$

$$
= \frac{1}{2}\mathbf{u}^T[T_{ij}]\mathbf{u} + \frac{1}{2}\mathbf{v}^T[T_{ij}]\mathbf{v} - \mathbf{u}^T[D_{ij}^X]\mathbf{p} - \mathbf{v}^T[D_{ij}^Y]\mathbf{p} - \mathbf{u}^T\mathbf{f}^x - \mathbf{v}^T\mathbf{f}^y. \tag{89}
$$

We assume that the augmented matrix is positive definite so that a minimization of $f(\mathbf{u},\mathbf{v},\mathbf{p})$ has a unique solution. (This condition relies upon the relative magnitudes of δ and ν. As long as δ is sufficiently large, indicating a small time step, then this will be the case.) It was mentioned previously that we could substitute out **u** and **v** using Eqs. (74) making f solely a function of **p**, however, to avoid the algebraic clutter we shall refrain from doing so at present. Just remember that **u** and **v** are implicit functions of **p**. Consider the following iterative updating scheme for pressure $\mathbf{p}_m$

$$
\mathbf{p}_{m+1} = \mathbf{p}_m - \alpha_m \mathbf{d}_m \tag{90}
$$

where, as before, $\mathbf{d}_m$ is the current search direction and α_m is the current search length. From Eqs. (73) (with $\lambda=\mu=\mathbf{0}$) we find

$$
\begin{aligned}
\mathbf{u}_{m+1} &= [T_{ij}]^{-1}\mathbf{f}^x + [T_{ij}]^{-1}[D_{ij}^X]\mathbf{p}_{m+1} = [T_{ij}]^{-1}\mathbf{f}^x + [T_{ij}]^{-1}[D_{ij}^X]\mathbf{p}_m - \alpha_m[T_{ij}]^{-1}[D_{ij}^X]\mathbf{d}_m, \\
\mathbf{v}_{m+1} &= [T_{ij}]^{-1}\mathbf{f}^y + [T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p}_{m+1} = [T_{ij}]^{-1}\mathbf{f}^x + [T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p}_m - \alpha_m[T_{ij}]^{-1}[D_{ij}^Y]\mathbf{d}_m.
\end{aligned} \tag{91}
$$

Equations (90) and (91) constitute the update scheme for an iterative algorithm once we have decided upon the search directions $\mathbf{d}_m$. The search length α_m can be found, as before, by direction minimization of Eq. (89) with respect to α_m. To do so note that $\mathbf{p}_{m+1}$, $\mathbf{u}_{m+1}$, and $\mathbf{v}_{m+1}$ are all functions of α_m through Eqs. (90) and (91). Their derivatives with respect to α_m are given by

$$
\begin{aligned}
\frac{\partial \mathbf{p}_{m+1}}{\partial \alpha_m} &= -\mathbf{d}_m, \\
\frac{\partial \mathbf{u}_{m+1}}{\partial \alpha_m} &= -[T_{ij}]^{-1}[D_{ij}^X]\mathbf{d}_m, \\
\frac{\partial \mathbf{v}_{m+1}}{\partial \alpha_m} &= -[T_{ij}]^{-1}[D_{ij}^Y]\mathbf{d}_m.
\end{aligned} \tag{92}
$$

Defining the scalar function $\varphi(\alpha_m)$ as

$$
\phi(\alpha_m) \equiv f\left(\mathbf{p}_{m+1}(\alpha_m), \mathbf{u}_{m+1}(\alpha_m), \mathbf{v}_{m+1}(\alpha_m)\right) \tag{93}
$$

we take its derivative and set it equal to zero

$$
\frac{\partial \phi(\alpha_m)}{\partial \alpha_m} = \frac{\partial f\left(\mathbf{p}_{m+1},\mathbf{u}_{m+1},\mathbf{v}_{m+1}\right)}{\partial \mathbf{p}_{m+1}}\frac{\partial \mathbf{p}_{m+1}}{\partial \alpha_m} + \frac{\partial f\left(\mathbf{p}_{m+1},\mathbf{u}_{m+1},\mathbf{v}_{m+1}\right)}{\partial \mathbf{u}_{m+1}}\frac{\partial \mathbf{u}_{m+1}}{\partial \alpha_m} + \frac{\partial f\left(\mathbf{p}_{m+1},\mathbf{u}_{m+1},\mathbf{v}_{m+1})\right)}{\partial \mathbf{v}_{m+1}}\frac{\partial \mathbf{v}_{m+1}}{\partial \alpha_m} = 0. \tag{94}
$$

The above equation is solved for α_m with the result

$$\alpha_m = \frac{\mathbf{d}_m^T[\Delta_{ij}]\mathbf{p}_m + \mathbf{d}_m^T\left([D_{ij}^X]^T[T_{ij}]^{-1}\mathbf{f}^x + [D_{ij}^Y]^T[T_{ij}]^{-1}\mathbf{f}^y\right)}{\mathbf{d}_m^T[\Delta_{ij}]\mathbf{d}_m}. \qquad (95)$$

Note that this formula is not as convenient as the one generated for the simple system $[A_{ij}]\mathbf{x}=\mathbf{y}$. To use the above equation we must invert and store the matrix $[T_{ij}]$ and form the matrix $[\Delta_{ij}]$, a costly procedure considering the potential size of the system. However, the procedure would have to be done only once for each value of the parameter δ. Depending upon circumstance it might be faster simply to solve for the pressure directly using Eq. (88).

A simple interpretation results if the system is such that the drive vectors $\mathbf{f}^x$ and $\mathbf{f}^y$ are representable from a single vector $\mathbf{g}$ in the pressure domain according to

$$\begin{aligned} \mathbf{f}^x &= [D_{ij}^X]\mathbf{g}, \\ \mathbf{f}^y &= [D_{ij}^Y]\mathbf{g}. \end{aligned} \qquad (96)$$

Then the update formula for the pressure $\mathbf{p}_{m+1}$ simplifies to

$$\mathbf{p}_{m+1} = \mathbf{p}_m - \left\langle \bar{\mathbf{d}}_m, \mathbf{p}_m + \mathbf{g} \right\rangle_\Delta \bar{\mathbf{d}}_m \qquad (97)$$

where the notation is analogous to that described in (84). In this case the solution for pressure is $\mathbf{p}=-\mathbf{g}$ which comes directly from Eq. (78) and is implied by Eq. (97).

3.4.1.4 A Simple Iterative Algorithm for the Stokes System

Here we present a iterative algorithm which will solve the Navier-Stokes system for the case when $\lambda=\mu=0$. This is a simple "ad hoc" method where the results of the previous subsection are essentially ignored. That is, the choice of α_m is not guided by any direct test of the dependent variables. Instead, we are guided to this algorithm from the physics of the fluid-flow problem. This presentation shows that it is possible to implement very simple algorithms based on physical intuition (that are guaranteed to converge), however, the rate of convergence may not be practical.

This method chooses the search directions for $\mathbf{p}_m$ as the divergence of the velocity field. That is, we choose

$$\mathbf{d}_m = [D_{ij}^X]^T\mathbf{u}_m + [D_{ij}^Y]^T\mathbf{v}_m \qquad (98)$$

which is the discrete analogue of $d=\nabla\cdot\mathbf{U}$. The values of $\mathbf{p}_m$, $\mathbf{u}_m$, and $\mathbf{v}_m$ are then updated according to

$$\mathbf{p}_{m+1} = \mathbf{p}_m - \alpha\mathbf{d}_m \qquad (99)$$

and

$$\begin{aligned} \mathbf{u}_{m+1} &= [T_{ij}]^{-1}\mathbf{f}^x + [T_{ij}]^{-1}[D_{ij}^X]\mathbf{p}_{m+1}, \\ \mathbf{v}_{m+1} &= [T_{ij}]^{-1}\mathbf{f}^y + [T_{ij}]^{-1}[D_{ij}^Y]\mathbf{p}_{m+1}, \end{aligned} \qquad (100)$$

which are the same as Eqs. (90) and (91) except for the search length parameter α_m. Here all the search lengths α_m are chosen to be the same number, α, which lies in the interval $(0,2\nu/N)$. With these conditions the algorithm is guaranteed to converge [16]. To see that the convergent solution is indeed the correct solution note that the algorithm is essentially a fixed-point algorithm which converges when the (discrete) divergence of the flow is zero. Thus, at that time the search direction will be zero. The value of the fixed point (**p**,**u**,**v**) is then found from Eqs. (98) to (100)

$$\begin{aligned}\mathbf{p} &= \mathbf{p} - \alpha\mathbf{d} = \mathbf{p} - \alpha\left([D_{ij}^X]^T\mathbf{u} + [D_{ij}^Y]\mathbf{v}\right) \\ &= \mathbf{p} - \alpha\left([D_{ij}^X]^T[T_{ij}]^{-1}\mathbf{f}^x + [D_{ij}^Y][T_{ij}]^{-1}\mathbf{f}^y + [\Delta_{ij}]\mathbf{p}\right)\end{aligned} \tag{101}$$

whose solution is

$$\mathbf{p} = -[D_{ij}]^{-1}\left([D_{ij}^X]^T[T_{ij}]^{-1}\mathbf{f}^x + [D_{ij}^Y]^T[T_{ij}]^{-1}\mathbf{f}^y\right). \tag{102}$$

From Eq. (88) this is seen to be correct.

The above algorithm will be quite slow for flow at large Reynolds number (i.e., a small value of viscosity ν). To further aggravate this situation, the time step Δt must be small to follow the fast dynamics of these flows. Thus, to improve the speed of convergence we use a preconditioned conjugate gradient version of the above algorithm.

3.4.1.5 A Preconditioned Conjugate-Gradient Algorithm

Here we outline a conjugate-gradient algorithm with preconditioning which has been presented by Glowinski et. al. [14]. The method is initialized by picking (arbitrarily) vectors $\mathbf{p}_0$, λ_0, and μ_0. Then we solve the linear system

$$\begin{aligned}\delta[A_{ij}]\mathbf{u}_0 + \nu[B_{ij}]\mathbf{u}_0 &= \mathbf{f}^x + [D_{ij}^X]\mathbf{p}_0 + [E_{ij}]\boldsymbol{\lambda}_0 \\ \delta[A_{ij}]\mathbf{v}_0 + \nu[B_{ij}]\mathbf{v}_0 &= \mathbf{f}^y + [D_{ij}^Y]\mathbf{p}_0 + [E_{ij}]\boldsymbol{\mu}_0\end{aligned} \tag{103}$$

for $\mathbf{u}_0$ and $\mathbf{v}_0$. Now compute the initial "remainder" vectors

$$\begin{aligned}\mathbf{s}_0 &= [D_{ij}^X]^{\mathrm{T}}\mathbf{u}_0 + [D_{ij}^Y]^{\mathrm{T}}\mathbf{v}_0\,, \\ \mathbf{r}_0^x &= [E_{ij}]^{\mathrm{T}}\mathbf{u} - \mathbf{c}\,, \\ \mathbf{r}_0^y &= [E_{ij}]^{\mathrm{T}}\mathbf{v} - \mathbf{d}\,.\end{aligned} \tag{104}$$

To continue we must solve two auxiliary systems which may be identified as the "costate" equations for pressure and for the fictitious domain Lagrange multipliers. Beginning with the definitions

$$\boldsymbol{\phi} \equiv \left(\phi_i \,|\, i \in J\right), \quad \boldsymbol{\phi}^1 \equiv \left(\phi_i \,|\, i \in J_1\right), \quad \boldsymbol{\phi}^B \equiv \left(\phi_i \,|\, i \in J_B\right), \tag{105}$$

the discrete system

$$\begin{aligned}&[B_{ij}^{1}]\phi_0^1 - [N_{ij}]\phi_0^1 = \mathbf{s} - [B_{ij}^{B}]\phi_0^B \\ &\phi_0^B = 0\end{aligned} \tag{106}$$

where

$$\begin{aligned}
B_{ij}^{1} &= \int_{\Omega}\left[\frac{\partial G_i}{\partial x}\frac{\partial G_j}{\partial x} + \frac{\partial G_i}{\partial y}\frac{\partial G_j}{\partial y}\right]d\Omega \qquad \text{for } i,j \in J_1 \\
B_{ij}^{B} &= \int_{\Omega}\left[\frac{\partial G_i}{\partial x}\frac{\partial G_j}{\partial x} + \frac{\partial G_i}{\partial y}\frac{\partial G_j}{\partial y}\right]d\Omega \qquad \text{for } i \in J_1,\ j \in J_B \\
N_{ij} &= \int_{\Gamma_1}\left[\frac{\partial G_j}{\partial x}n_x + \frac{\partial G_j}{\partial y}n_y\right]G_i\, d\Gamma \qquad \text{for } i,j \in J_1
\end{aligned} \tag{107}$$

constitutes the pressure costate and must be solved for φ_0. Note that φ_0 is solved in the pressure space $P^h(\Omega)$. The above system is the discretization of the weak form of the Poisson equation

$$\begin{aligned}
-\nabla^2\phi &= \nabla\cdot\mathbf{U} && \text{in } \Omega \\
\frac{\partial\phi}{\partial\mathbf{n}} &= 0 && \text{on } \Gamma_0 \\
\phi &= 0 && \text{on } \Gamma_1
\end{aligned} \tag{108}$$

which typically occurs somewhere in the formulation of incompressible fluid simulations. The second auxiliary system to be solved is the costate system for the Lagrange multipliers λ and μ. In this system we solve

$$[M_{ij}^{x}]\mathbf{g}_0^x = \mathbf{r}_0^x \qquad \text{and} \qquad [M_{ij}^{y}]\mathbf{g}_0^y = \mathbf{r}_0^y \tag{109}$$

for the vectors $\mathbf{g}_0^x$ and $\mathbf{g}_0^y$. System (109) is the discrete version of the equation

$$a(\mathbf{g},\mathbf{b}) = \int_{\Gamma_c}\mathbf{r}\cdot\mathbf{b}\, d\Gamma \qquad \forall\, \mathbf{b} \in B(\Gamma_c) \tag{110}$$

where $a(\cdot,\cdot)$ is some bilinear form on $B(\Gamma_c)$. We shall choose $a(\cdot,\cdot)$ to be the standard inner product so that the matrices $[M_{ij}^X]$ and $[M_{ij}^Y]$ are simply the identity.

From the solutions to the two auxiliary systems we form the initial "gradient" vector $\mathbf{g}_0 \equiv (\mathbf{g}_0^p, \mathbf{g}_0^x, \mathbf{g}_0^y)$ (corresponding to $\mathbf{p}_0$, λ_0, and μ_0) according to

$$\mathbf{g}_0 \equiv \begin{pmatrix}\mathbf{g}_0^p \\ \mathbf{g}_0^x \\ \mathbf{g}_0^y\end{pmatrix} = \begin{pmatrix}\delta_0 + \nu\mathbf{s}_0 \\ \mathbf{r}_0^x \\ \mathbf{r}_0^y\end{pmatrix}. \tag{111}$$

Finally, we set our initial (negative) search directions $\mathbf{w}_0 \equiv (\mathbf{w}_0^p, \mathbf{w}_0^x, \mathbf{w}_0^y)$ equal to the gradient vectors $(\mathbf{g}_0^p, \mathbf{g}_0^x, \mathbf{g}_0^y)$ or

$$\mathbf{w}_0 \equiv \begin{pmatrix} \mathbf{w}_0^p \\ \mathbf{w}_0^x \\ \mathbf{w}_0^y \end{pmatrix} = \begin{pmatrix} \mathbf{g}_0^p \\ \mathbf{g}_0^x \\ \mathbf{g}_0^y \end{pmatrix} \qquad (112)$$

This completes the initialization of the algorithm.

For the main loop of the algorithm we assume that the iteration step $m \geq 0$ and that $\mathbf{p}_m$, λ_m, μ_m, $\mathbf{u}_m$, $\mathbf{v}_m$, $\mathbf{g}_m$, and $\mathbf{w}_m$ are all known. To find the next solution iterates $\mathbf{p}_{m+1}$, λ_{m+1}, μ_{m+1}, $\mathbf{u}_{m+1}$, $\mathbf{v}_{m+1}$, $\mathbf{g}_{m+1}$, and $\mathbf{w}_{m+1}$ we proceed as follows:

Solve the linear system

$$\begin{aligned} \delta[A_{ij}]\mathbf{u} + \nu[B_{ij}]\mathbf{u} &= [D_{ij}^X]\mathbf{w}_m^p + [E_{ij}]\mathbf{w}_m^x \\ \delta[A_{ij}]\mathbf{v} + \nu[B_{ij}]\mathbf{v} &= [D_{ij}^Y]\mathbf{w}_m^p + [E_{ij}]\mathbf{w}_m^y \end{aligned} \qquad (113)$$

for the interim velocities **u** and **v**. Now form the interim remainder vectors **s**, $\mathbf{r}^x$, and $\mathbf{r}^y$

$$\begin{aligned} \mathbf{s} &= [D_{ij}^X]^{\mathrm{T}}\mathbf{u} + [D_{ij}^Y]^{\mathrm{T}}\mathbf{v}, \\ \mathbf{r}^x &= [E_{ij}]^{\mathrm{T}}\mathbf{u}, \\ \mathbf{r}^y &= [E_{ij}]^{\mathrm{T}}\mathbf{v}. \end{aligned} \qquad (114)$$

and set the gradients

$$\mathbf{g}_m^p = \delta\boldsymbol{\phi} + \nu\mathbf{s}, \quad \mathbf{g}_m^x = \mathbf{r}^x, \quad \mathbf{g}_m^y = \mathbf{r}^y, \qquad (115)$$

where φ solves the system

$$\begin{aligned} [B_{ij}^1]\boldsymbol{\phi}^1 - [N_{ij}]\boldsymbol{\phi}^1 &= \mathbf{s} - [B_{ij}^B]\boldsymbol{\phi}^B \\ \boldsymbol{\phi}^B &= 0 \end{aligned} \qquad (116)$$

(the definitions for φ^1 and φ^B are as before). Next compute the *search length* parameter ρ_m according to

$$\rho_m = \frac{(\mathbf{r}_m^p)^T[A_{ij}^P]\mathbf{g}_m^p + (\mathbf{r}_m^x)^T[A_{ij}^B]\mathbf{g}_m^x + (\mathbf{r}_m^y)^T[A_{ij}^B]\mathbf{g}_m^y}{(\mathbf{r}^p)^T[A_{ij}^P]\mathbf{w}_m^p + (\mathbf{r}^x)^T[A_{ij}^B]\mathbf{w}_m^x + (\mathbf{r}^y)^T[A_{ij}^B]\mathbf{w}_m^y} \qquad (117)$$

where the matrix entries are given by

$$A_{ij}^P = \int_\Omega G_i G_j \, d\Omega \quad \text{for } i,j \in J \qquad \text{and} \qquad A_{ij}^B = \int_{\Gamma_c} H_i H_j \, d\Gamma \quad \text{for } i,j \in K. \qquad (118)$$

Update the vectors using the search directions, previous iterates, and search length parameter.

$$
\begin{aligned}
\mathbf{p}_{m+1} &= \mathbf{p}_m - \rho_m \mathbf{w}_m^p \\
\boldsymbol{\lambda}_{m+1} &= \boldsymbol{\lambda}_m - \rho_m \mathbf{w}_m^x \\
\boldsymbol{\mu}_{m+1} &= \boldsymbol{\mu}_m - \rho_m \mathbf{w}_m^y \\
\mathbf{u}_{m+1} &= \mathbf{u}_m - \rho_m \mathbf{u}_m \\
\mathbf{v}_{m+1} &= \mathbf{v}_m - \rho_m \mathbf{v}_m \\
\mathbf{g}_{m+1} &= \mathbf{g}_m - \rho_m \mathbf{g}_m
\end{aligned}
\tag{119}
$$

Finally, if

$$
\frac{(\mathbf{r}_{m+1}^p)^T [A_{ij}^P] \mathbf{g}_{m+1}^p + (\mathbf{r}_{m+1}^x)^T [A_{ij}^B] \mathbf{g}_{m+1}^x + (\mathbf{r}_{m+1}^y)^T [A_{ij}^B] \mathbf{g}_{m+1}^y}{(\mathbf{r}_0^p)^T [A_{ij}^P] \mathbf{g}_0^p + (\mathbf{r}_0^x)^T [A_{ij}^B] \mathbf{g}_0^x + (\mathbf{r}_0^y)^T [A_{ij}^B] \mathbf{g}_0^y} \le \varepsilon \tag{120}
$$

where ε is some solution tolerance parameter, then stop the algorithm and take the last iterate as the solution to the Stokes problem. Otherwise, compute the *search direction* parameter

$$
\gamma_m = \frac{(\mathbf{r}_{m+1}^p)^T [A_{ij}^P] \mathbf{g}_{m+1}^p + (\mathbf{r}_{m+1}^x)^T [A_{ij}^B] \mathbf{g}_{m+1}^x + (\mathbf{r}_{m+1}^y)^T [A_{ij}^B] \mathbf{g}_{m+1}^y}{(\mathbf{r}_m^p)^T [A_{ij}^P] \mathbf{g}_m^p + (\mathbf{r}_m^x)^T [A_{ij}^B] \mathbf{g}_m^x + (\mathbf{r}_m^y)^T [A_{ij}^B] \mathbf{g}_m^y} \tag{121}
$$

then update the search direction $\mathbf{w}_m$ to

$$
\mathbf{w}_{m+1} = \mathbf{g}_m + \gamma_m \mathbf{w}_m . \tag{122}
$$

Set the iteration counter to $m=m+1$ then restart the algorithm from Eq (113).

3.4.2 Solution of the Advection Problem

The discrete advection problem may be written

$$
\begin{aligned}
\delta[A_{ij}]\mathbf{u} + \nu[B_{ij}]\mathbf{u} + \left([C_{ijk}^X]\mathbf{u}\right)\mathbf{u} + \left([C_{ijk}^Y]\mathbf{v}\right)\mathbf{u} &= \mathbf{f}^{\mathbf{x}} \\
\delta[A_{ij}]\mathbf{v} + \nu[B_{ij}]\mathbf{v} + \left([C_{ijk}^X]\mathbf{u}\right)\mathbf{v} + \left([C_{ijk}^Y]\mathbf{v}\right)\mathbf{v} &= \mathbf{f}^{\mathbf{y}}
\end{aligned}
\tag{123}
$$

where, again, δ is the reciprocal of the time step and $\mathbf{f}^x$ and $\mathbf{f}^y$ have been generalized to include the entire right-hand side of Eq. (69). This is a nonlinear system due to the terms involving the tensors $[C_{ijk}^X]$ and $[C_{ijk}^Y]$, however, there are no constraints and the pressure $\mathbf{p}$ and Lagrange multipliers λ and μ do not appear. This is set of second-degree algebraic equations in $\mathbf{u}$ and $\mathbf{v}$ where there are as many equations as there are unknowns. Therefore, we can use standard techniques for solving this nonlinear set of equations. In particular Newton's method is readily applicable. Yet, it is also possible to use other iterative techniques such as a nonlinear conjugate gradient method. To do so, however, a least-squares formulation of Eqs. (123) must be provided.

3.4.2.1 A Functional Equation for the Advection System

A functional equation for the advection problem may be written just as was done for the Stokes problem. Presumably then, minimizing the functional with respect to $\mathbf{u}$ and $\mathbf{v}$ will result in the solution to the advection problem. However, in this case the functional is of

third order and not necessarily convex. Thus, the minimization will require more sophisticated techniques. Proceeding analogously as in Section 3.4.1.3, the resulting functional f is

$$f(\mathbf{u},\mathbf{v}) = \mathbf{u}^T[T_{ij}]\mathbf{u} + \mathbf{v}^T[T_{ij}]\mathbf{v} + \mathbf{u}^T\left[\left([C_{ijk}^X]\mathbf{u}\right)\mathbf{u} + \left([C_{ijk}^Y]\mathbf{v}\right)\right]\mathbf{u} + \mathbf{v}^T\left[\left([C_{ijk}^X]\mathbf{u}\right)\mathbf{u} + \left([C_{ijk}^Y]\mathbf{v}\right)\right]\mathbf{v} - \mathbf{u}^T\mathbf{f}^x - \mathbf{v}^T\mathbf{f}^y \tag{124}$$

where the matrix $[T_{ij}]$ is as before (defined by Eq. (72)). As of yet we have not pursued this solution strategy. Instead, we have relied upon a linearized approximation of the advection system. It may be necessary to solve the full nonlinear system if we encounter any accuracy problems in the future.

3.4.2.2 A Linearized Approximation

Currently, we solve this problem using a linearized approximation to the second-degree terms in the advection equations. We do this by using the previous iterates of **u** and **v** for the initial multiplications of the advection tensors, that is

$$\left([C_{ijk}^X]\mathbf{u}_{n+1-\theta}^0\right)\mathbf{u}_{n+1-\theta}^0 + \left([C_{ijk}^Y]\mathbf{v}_{n+1-\theta}^0\right)\mathbf{u}_{n+1-\theta}^0 \approx \left([C_{ijk}^X]\mathbf{u}_{n+\theta}^0\right)\mathbf{u}_{n+1-\theta}^0 + \left([C_{ijk}^Y]\mathbf{v}_{n+\theta}^0\right)\mathbf{u}_{n+1-\theta}^0,$$
$$\text{and} \tag{125}$$
$$\left([C_{ijk}^X]\mathbf{u}_{n+1-\theta}^0\right)\mathbf{v}_{n+1-\theta}^0 + \left([C_{ijk}^Y]\mathbf{v}_{n+1-\theta}^0\right)\mathbf{v}_{n+1-\theta}^0 \approx \left([C_{ijk}^X]\mathbf{u}_{n+\theta}^0\right)\mathbf{v}_{n+1-\theta}^0 + \left([C_{ijk}^Y]\mathbf{v}_{n+\theta}^0\right)\mathbf{v}_{n+1-\theta}^0.$$

It has been noted that there is practically no loss in accuracy and stability when using this approximation [13]. The resulting linear system is substantially smaller than the Stokes system. Consequently, using direct techniques to solve this system has not produced any significant bottleneck concerning real-time computation.

4 Control Problem Structure for Fluid Flow

We have been investigating the control problem structure for fluid flow in both the discrete (FEM/FDM) and the continuous case. In the discrete case our system is governed by a set of nonlinear, first-order, ordinary differential equations with constraints. We consider aspects and techniques of controlling these equations directly. However, this approach does not necessarily provide us with a clear technique for controlling the actual physical system. Thus, we are also investigating the boundary layer control of the continuous, incompressible Navier-Stokes equations.

4.1 The Discrete Case

The control theory for the discrete Navier-Stokes equations is in the very earliest stages of development. Our approach has been influenced by the important work of Glowinski, Temam, Lions, Banks, and others. However, for the most part this work has not led to "practical" feedback laws that can be tested in simulations or in the laboratory. Since this is our primary goal, our approach has accordingly been somewhat pragmatic – find control laws based on measurable quantities that can be implemented with reasonable actuation schemes and evaluated in simulation. We have limited ourselves to finite dimensional measurements, and to actuators with finite dimensional influence functions. Thus, we assume that the flow state can be measured only at a finite number of points (near or on

the boundary of the object in the flow). And we assume that the actuators act at a finite number of points.

To simplify the presentation, consider the numerical representation of the system dynamics in the following form:

$$
\begin{aligned}
&[A]\dot{U} + \left([[B]]U\right)U + [C]U + [D]P = F + [Q]\Lambda \\
&[D^T]U = 0 \\
&[Q^T]U = X
\end{aligned}
$$

where $[A]$, $[C]$, $[D]$ and $[Q]$ are matrices and $[[B]]$ is a rank-three tensor. The quantity U is the discrete vector representing the velocity field, the vector P is the pressure, the vector Λ is the fictitious domain Lagrange multiplier for an embedded object, the vector F is the applied force, and vector X represent the boundary control. Since the velocity and pressure grids may be different, the dimensions of U and P are not necessarily related. The first equation is the discrete approximation of the momentum equations. The last two equations represent the divergence condition and the boundary control condition on the object. In effect, this is a kind of differential-algebraic system – a control system with an algebraic constraint. There are several ways to handle such systems. Our approach is based on singular perturbation methods (suggested by the numerical methods). We introduce additional state variables and re-write the equations in the form

$$
\begin{aligned}
&[A]\dot{U} + \left([[B]]U\right)U + [C]U + [D]P = F + [Q]\Lambda \\
&\mu\dot{\xi} = \alpha\xi + [D^T]U \\
&\varepsilon\dot{\zeta} = \beta\zeta - [Q^T]U + X
\end{aligned}
$$

Here (ε,μ) are small (positive) parameters, (α,β) are matrices having eigenvalues with negative real parts, and (ξ,ζ) are state variables of appropriate dimensions. The state ζ represents the "dynamics" of the boundary control system (as a dynamical compensator), and the state ξ is an artifice introduced to approximate the divergence condition.[1] The parameters (ε,μ) are "small" indicating our intent to use (singular) perturbation methods; but also indicating the physical interpretation that the dynamics of the boundary control system are "fast" relative to the (large scale) dynamics of the fluid flow. In effect, the flow conditions near the boundary change slowly (in time) relative to the capability of the control system actuator dynamics. With this formulation, several options are available for control. Suppose we want to control the flow to stabilize the velocity field U in the presence of disturbances F acting in the flow. In this case, the control objective is to choose the (boundary) control values $X(t)$ to "reject" the disturbances and stabilize the flow. In the absence of the nonlinear term in the above, this is an elementary problem in linear control theory. Since disturbance rejection and adaptive stabilization methods have been available for some time in nonlinear control theory, tools are available in principle to resolve the stabilization control problem for the Navier-Stokes equation.

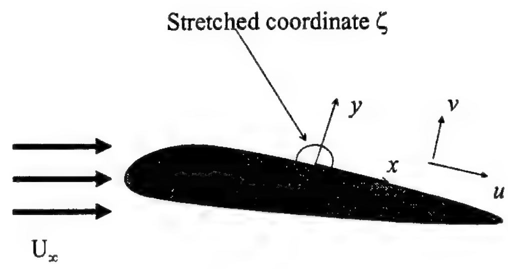
4.2 The Continuous Case

In this situation we consider the continuous Navier-Stokes equations directly in order to develop some physical insight into the controls problem. Perturbation methods are

[1] The divergence condition state is not necessary, we can use differential – algebraic methods; it is used here to avoid the complexities of this condition. In control theory parlance, this condition (without the state) amounts to sliding mode constraint.

employed to derive approximate differential systems that describe the behavior of the boundary layer around objects in the fluid domain. In the spirit of Prandtl we proceed using a boundary layer expansion of the Navier-Stokes equations, however, we wish to retain the time dependent terms. In order to do so we find that a multiple time scale expansion may be used to account for changes in the normal fluid velocity.

Using the principles of differential geometry, we have written a Mathematica program to symbolically compute the incompressible Navier-Stokes equations in arbitrary coordinate systems (see Appendix A). For example, consider the airfoil depicted in Figure 9 embedded in the 2D fluid domain. Construct a coordinate system around it so that x is the tangential distance along its surface and y is the distance in direction normal to the surface, as shown in the figure. (Since we are going to be interested only in the boundary layer, y is defined in the layer even if the airfoil is not locally convex.) The surface of the airfoil is described by the curve y=0.



• Figure 9: Airfoil with stretched local coordinates

By applying an asymptotic boundary layer expansion $\zeta=(Re)y$ in the normal coordinate (i.e., a stretching in the y direction) then taking the limit as the Reynolds number Re goes to infinity we get the following equations for the dynamics of the boundary layer:

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial \zeta} + \frac{\partial p}{\partial x} &= \frac{\partial^2 u}{\partial \zeta^2} \\ \frac{\partial p}{\partial \zeta} &= \frac{1}{\sqrt{Re}}\kappa(x)u^2 \\ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial \zeta} &= 0 \end{aligned} \tag{126}$$

where u is the velocity in the x direction, v is the velocity in the ζ direction (y direction), p is the pressure, and $\kappa(x)$ is the curvature of the surface at x (in the 2D case it is simply a scalar, in 3D it is a rank-two tensor). Notice that these are the Prandtl limit equations for the flat plate with the addition of the term accounting for surface curvature. The boundary conditions are given by

$$\begin{aligned} &u(x,0;t) = 0 & &u(x,\zeta;t) \xrightarrow[\zeta\to\infty]{} u_e(x,0) \\ &v(x,0;t) = X(t) & &v(x,\zeta;t) \xrightarrow[\zeta\to\infty]{} v_e(x,0) \\ & & &p(x,\zeta;t) \xrightarrow[\zeta\to\infty]{} p_e(x,0) \end{aligned} \tag{127}$$

where X(t) is the boundary control (which may be nonzero only on portions of the surface), and u_e, v_e, and p_e are the inviscid solutions to the problem (i.e., when $Re\to\infty$) in the (x,y) coordinates, the so-called outer solution. We are currently investigating these equations

using asymptotic and perturbation analysis along with numerical methods to develop control strategies. One point that is immediately obvious from the second equation in (126) is that pressure gradient is highly sensitive to transverse velocity. Thus, as we know empirically, at points of flow separation there are very large pressure differentials. For example, for flow over corners $\kappa(x)$ becomes very large and, since u must remain nonzero by Newton's first law (causing separation), the pressure gradient is also quite large. Note that Eqs. (126) involve the tangential velocity u, the normal velocity v, and the curvature κ. We have been attempting to use the normal velocity at the surface as a control parameter. Another possibility is to use the curvature κ as the control, as in compliant membranes or flap actuators.

One final point, if we introduce a fast time-scale variable $\tau \equiv (Re)t$, we can recover the v time dependence in the second boundary layer equation

$$\frac{\partial v}{\partial \tau}+\frac{\partial p}{\partial \zeta}=\frac{1}{\sqrt{Re}}\kappa(x)u^2 \qquad (128)$$

This is an important equation when considering control strategies. It can be shown that $\partial u/\partial \tau=0$ (to first order) so that u may be considered constant in the above equation. Thus, changes in normal velocity v and pressure p can happen on very fast time and spatial scales. This equation can account for boundary layer "bursting" seen in experiment.

4.2.1 Boundary Layer Analysis (Singular Perturbation)

Here we demonstrate the validity of Eqs. (126) and (127). Rather than developing these relations using differential geometry as in the Mathematica code, we use a more direct boundary layer approach to clarify the ideas. The results of the differential analysis will simply add an additional term accounting for surface curvature. We start with the two-dimensional, incompressible Navier-Stokes equations in cartesian coordinates.

$$\begin{gathered}
\dot{u}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+\frac{\partial p}{\partial x}=\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)\\
\dot{v}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+\frac{\partial p}{\partial y}=\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)\\
\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0
\end{gathered} \qquad (129)$$

where the over dot indicates differentiation with respect to time, u and v are the fluid velocities in the x and y direction, respectively, p is the pressure, and ν is the kinematic viscosity (equal to $1/Re$). Of course there will be boundary conditions that accompany the above system; these conditions will depend upon the fluid domain and the particular material composing the boundaries. We can represent these boundary conditions symbolically according to the following:

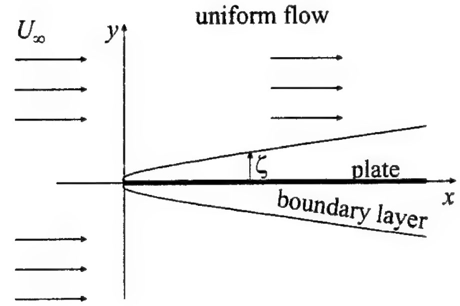
$$B_i\left[u(x,y;t),v(x,y;t),p(x,y;t)\right]=\beta_i \quad \text{for } i=1,2,\ldots,N_B \qquad (130)$$

where the $B[\cdot,\cdot,\cdot]$ are the boundary condition operators and N_B is the number of boundary conditions. Because of the time derivatives we also have initial conditions associated with the time-dependent Navier-Stokes equations. That is, we require an initial state in time for the fluid that can be represented as

$$
\begin{aligned}
u(x,y;t)\,|_{t=0} &= u_i(x,y),\\
v(x,y;t)\,|_{t=0} &= v_i(x,y),\\
p(x,y;t)\,|_{t=0} &= p_i(x,y),
\end{aligned}
\tag{131}
$$

where u_i, v_i and p_i are the initial states of the field variables.

We are interested in flows with large Reynolds number *Re*, thus, we proceed by considering the limiting case where $\nu \to 0$. For the moment, consider the problem of a flat plate immersed in a uniform flow. The plate is infinitesimally thin, aligned in the plane $y=0$, and has no-slip boundary conditions. This situation is depicted graphically in Figure 10. When $\nu=0$ we have the Euler equations whose solution, in this case, is simple uniform flow which violates the no-slip boundary conditions. Prandtl observed that in order for a limiting solution to make sense there must be a "boundary layer" around the plate where the flow changes rapidly from zero to its uniform flow velocity [6]. Thus, we expect a rapid transition in u and v around the positions $y=0$. This idea is further supported by the fact that the viscosity ν multiplies the highest order derivatives in the above equations, suggesting that these derivatives must approach infinity in the limit $\nu \to 0$ in order to satisfy the boundary conditions.



• Figure 10: Uniform flow around a flat plate

In what follows we are going to compute an asymptotic expansion of the Navier-Stokes equations in terms of the viscosity ν. The solutions to the expanded system will approximate fluid flow for small viscosities. Due to the boundary layer around the plate (or any object immersed in fluid flow) it is necessary to expand the equations in two separate regions. One expansion is valid in the boundary layer and is referred to as the *inner expansion*. The other expansion is valid in the region outside the boundary layer and is referred to as the *outer expansion*. Once the solutions in the opposing regions are found, they must be matched together to produce the full composite expansion. This matching process can be tricky when consider higher order terms.

We continue to focus on the flat plate problem, so that we assume the existence of a boundary layer around the points $y=0$. We consider the case of wall curvature later. The boundary conditions for a flat plate in uniform flow are

$$
\begin{matrix} u(x,0;t)=0 \\ v(x,0;t)=0 \end{matrix}\quad \text{for } x>0 \qquad \begin{matrix} u(x,y;t)=U_\infty \\ v(x,y;t)=0 \end{matrix}\quad \text{for } x<0 \qquad \begin{matrix} \lim\limits_{y\to\infty} u(x,y;t)=U_\infty \\ \lim\limits_{y\to\infty} v(x,y;t)=0 \\ \lim\limits_{y\to\infty} p(x,y;t)=P_\infty \end{matrix} \tag{132}
$$

where U_∞ and P_∞ are the free stream flow velocity and pressure, respectively.

4.2.1.1 The Outer Expansion

In the outer expansion equations the field variables are denoted with capital letters, thus, we have *U*, *V*, and *P* as the outer solutions for *x*-velocity, *y*-velocity, and pressure, respectively. To begin with assume that the field variables can be expanded in a power series in ν

$$
\begin{aligned}
U(x,y;t) &= U_0(x,y;t) + \nu U_1(x,y;t) + O(\nu^2) \\
V(x,y;t) &= V_0(x,y;t) + \nu V_1(x,y;t) + O(\nu^2) \\
P(x,y;t) &= P_0(x,y;t) + \nu P_1(x,y;t) + O(\nu^2)
\end{aligned}
\tag{133}
$$

Substituting these expansions into the Navier-Stokes equations and collecting terms with like powers of ν yields separate systems of equations and boundary conditions for each order of ν. The zero-order system, $O(1)$, is given by

$$
O(1) \qquad
\begin{aligned}
\dot{U}_0 + U_0\frac{\partial U_0}{\partial x} + V_0\frac{\partial U_0}{\partial y} + \frac{\partial P_0}{\partial x} &= 0 \\
\dot{V}_0 + U_0\frac{\partial V_0}{\partial x} + V_0\frac{\partial V_0}{\partial y} + \frac{\partial P_0}{\partial y} &= 0 \\
\frac{\partial U_0}{\partial x} + \frac{\partial V_0}{\partial y} &= 0
\end{aligned}
\tag{134}
$$

$$
O(1) \qquad B_i\left[U_0(x,y;t), V_0(x,y;t), P_0(x,y;t)\right] = \beta_i \quad \text{for condition at } y \to \infty \tag{135}
$$

$$
O(1) \qquad
\begin{aligned}
U_0(x,y;t)\,|_{t=0} &= u_i(x,y), \\
V_0(x,y;t)\,|_{t=0} &= v_i(x,y), \\
P_0(x,y;t)\,|_{t=0} &= p_i(x,y),
\end{aligned}
\tag{136}
$$

while the first-order system, O(ν), is given by

$$
O(\nu) \qquad
\begin{aligned}
\dot{U}_1 + U_0\frac{\partial U_1}{\partial x} + U_1\frac{\partial U_0}{\partial x} + V_0\frac{\partial U_1}{\partial y} + V_1\frac{\partial U_0}{\partial y} + \frac{\partial P_1}{\partial x} &= \frac{\partial^2 U_0}{\partial x^2} + \frac{\partial^2 U_0}{\partial y^2} \\
\dot{V}_1 + U_0\frac{\partial V_1}{\partial x} + U_1\frac{\partial V_0}{\partial x} + V_0\frac{\partial V_1}{\partial y} + V_1\frac{\partial V_0}{\partial y} + \frac{\partial P_1}{\partial y} &= \frac{\partial^2 V_0}{\partial x^2} + \frac{\partial^2 V_0}{\partial y^2} \\
\frac{\partial U_1}{\partial x} + \frac{\partial V_1}{\partial y} &= 0
\end{aligned}
\tag{137}
$$

$$
O(\nu) \qquad B_i\left[U_1(x,y;t), V_1(x,y;t), P_1(x,y;t)\right] = 0 \quad \text{for condition at } y \to \infty \tag{138}
$$

$$
O(\nu) \qquad
\begin{aligned}
U_1(x,y;t)\,|_{t=0} &= 0, \\
V_1(x,y;t)\,|_{t=0} &= 0, \\
P_1(x,y;t)\,|_{t=0} &= 0,
\end{aligned}
\tag{139}
$$

The zero-order equations are the Euler equations exactly. Consequently, it is common to denote the field variable solutions with a subscript *e*. That is our variables U_0, V_0, and P_0 also appear as u_e, v_e, and p_e, respectively, in the literature. Note that once the zero-order (Euler) system is solved for U_0, V_0, and P_0, the first-order system is linear in U_1, V_1, and P_1.

Since the expansion of (133) must hold for all ν, the initial conditions must be satisfied by the zero-order system while the first-order system will have zero initial conditions. This fact also dictates the form of the boundary conditions on future expanded systems.

4.2.1.2 The Inner Expansion (Boundary Layer)

We shall consistently use lower case letter to indicate field variables in the inner expansion. As we shall see, there will be several different sets of variables resulting from the application of multiple transforms.

In boundary layer the field variables experience extreme variation. Thus, we expect a rapid variation with the coordinate *y* normal to the plate around the points $y=0$ (for $x>0$). This condition suggest the stretching transform for the normal coordinate *y*

$$\zeta \equiv \frac{y}{\nu^\alpha} \qquad \Rightarrow \qquad \frac{\partial}{\partial y} = \frac{1}{\nu^\alpha}\frac{\partial}{\partial \zeta} \qquad (140)$$

where α is some positive number to be determined. By converting to the coordinate ζ the boundary layer can be held to a nonzero thickness as the viscosity ν approaches zero. Because the terms of the Navier-Stokes equations that contain the second-order derivative $\partial^2/\partial\zeta^2$ have a factor ν, the proper choice for α is known to be

$$\alpha = \frac{1}{2} \qquad (141)$$

If it is taken larger,we ecounter an unbounded limit, if taken smaller the limit equations fails to generate any new information.

The divergence condition in the governing equations implies the existence of a stream function $\psi(x,y)$ such that

$$u(x,y) = \frac{\partial\psi(x,y)}{\partial y} = \frac{1}{\nu^{1/2}}\frac{\partial\psi(x,\nu^{1/2}\zeta)}{\partial\zeta}, \qquad v(x,y) = -\frac{\partial\psi(x,y)}{\partial x} = -\frac{\partial\psi(x,\nu^{1/2}\zeta)}{\partial x}. \qquad (142)$$

Since we expect $\partial\psi(x,y)/\partial\zeta>0$, we must use the following transformations to keep *u* bounded as $\nu\to 0$:

$$\begin{aligned}
\psi'(x,\zeta) &\equiv \frac{1}{\nu^{1/2}}\psi(x,\nu^{1/2}\zeta), \\
u'(x,\zeta) &\equiv u(x,\nu^{1/2}\zeta), \\
v'(x,\zeta) &\equiv \frac{1}{\nu^{1/2}}v(x,\nu^{1/2}\zeta), \\
p'(x,\zeta) &\equiv p(x,\nu^{1/2}\zeta),
\end{aligned} \qquad (143)$$

where the last three transforms follow from the first. Applying the stretching transform to the Navier-Stokes equations along with the above transformations for the field variables yields the new system

$$
\begin{gathered}
\dot{u}'+u'\frac{\partial u'}{\partial x}+v'\frac{\partial u'}{\partial \zeta}+\frac{\partial p'}{\partial x}=\nu\frac{\partial^2 u'}{\partial x^2}+\frac{\partial^2 u'}{\partial \zeta^2} \\
\nu\left(\dot{v}'+u'\frac{\partial v'}{\partial x}+v'\frac{\partial v'}{\partial \zeta}\right)+\frac{\partial p'}{\partial \zeta}=\nu^2\frac{\partial^2 v'}{\partial x^2}+\nu\frac{\partial^2 v'}{\partial \zeta^2} \\
\frac{\partial u'}{\partial x}+\frac{\partial v'}{\partial \zeta}=0
\end{gathered}
\tag{144}
$$

where the second equation has been multiplied by $\nu^{1/2}$. Assuming that the transformed field variables can be expanded according to

$$
\begin{aligned}
u'(x,\zeta;t)&=u_0(x,\zeta;t)+\nu u_1(x,\zeta;t)+O(\nu^2),\\
v'(x,\zeta;t)&=v_0(x,\zeta;t)+\nu v_1(x,\zeta;t)+O(\nu^2),\\
p'(x,\zeta;t)&=p_0(x,\zeta;t)+\nu p_1(x,\zeta;t)+O(\nu^2),
\end{aligned}
\tag{145}
$$

the equations for the components are found by collecting terms with the same power of ν as a factor. The zero-order component system is found to be

$$
O(1)\qquad
\begin{gathered}
\dot{u}_0+u_0\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_0}{\partial x}=\frac{\partial^2 u_0}{\partial \zeta^2} \\
\frac{\partial p_0}{\partial \zeta}=0 \\
\frac{\partial u_0}{\partial x}+\frac{\partial v_0}{\partial \zeta}=0
\end{gathered}
\tag{146}
$$

$$
O(1)\qquad B_i\left[u_0(x,\zeta;t),v_0(x,\zeta;t),p_0(x.\zeta;t)\right]=\beta_i \quad \text{for condition at } y=\zeta=0
\tag{147}
$$

$$
O(1)\qquad
\begin{aligned}
&u_0(x,\zeta;t)|_{t=0}=u_i(x,\nu^{1/2}\zeta),\\
&v_0(x,\zeta;t)|_{t=0}=\nu^{1/2}v_i(x,\nu^{1/2}\zeta),\\
&p_0(x,\zeta;t)|_{t=0}=p_i(x,\nu^{1/2}\zeta),
\end{aligned}
\tag{148}
$$

The equations for the first-order system are

$$
O(\nu)\qquad
\begin{gathered}
\dot{u}_1+u_0\frac{\partial u_1}{\partial x}+u_1\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_1}{\partial \zeta}+v_1\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_1}{\partial x}=\frac{\partial^2 u_1}{\partial \zeta^2}+\frac{\partial^2 u_0}{\partial x^2} \\
\dot{v}_0+u_0\frac{\partial v_0}{\partial x}+v_0\frac{\partial v_0}{\partial \zeta}+\frac{\partial p_1}{\partial \zeta}=\frac{\partial^2 v_0}{\partial \zeta^2} \\
\frac{\partial u_1}{\partial x}+\frac{\partial v_1}{\partial \zeta}=0
\end{gathered}
\tag{149}
$$

$O(\nu)$ $\quad B_i[u_1(x,\zeta;t), v_1(x,\zeta;t), p_1(x,\zeta;t)] = 0$ for condition at $y = \zeta = 0$ (150)

$$u_1(x,\zeta;t)|_{t=0} = 0,$$

$O(\nu)$ $\quad v_1(x,\zeta;t)|_{t=0} = 0,$ (151)

$$p_1(x,\zeta;t)|_{t=0} = 0,$$

The zero-order system is the Prandtl boundary layer equations with the addition of a time varying term (these equations are also verified by the Mathematica program). They are completely independent, that is, they only involve the zero-order field variables. However, the first-order system is coupled to the zero-order system. Yet, unlike the zero-order system, the first-order system is linear. Assuming that the zero-order system can be solved, the solution can be inserted into Eq. (149) to yield a linear system for the first-order field variables.

We note that the dynamics equations for the above system may be expressed in an alternate form after some manipulation. The alternate forms are as follows:

$$O(1) \qquad \begin{gathered} \dot{u}_0 + \frac{\partial}{\partial x}\left(u_0^{\,2}\right) + \frac{\partial}{\partial \zeta}\left(u_0 v_0\right) + \frac{\partial p_0}{\partial x} = \frac{\partial^2 u_0}{\partial \zeta^2} \\ \frac{\partial p_0}{\partial \zeta} = 0 \\ \frac{\partial u_0}{\partial x} + \frac{\partial v_0}{\partial \zeta} = 0 \end{gathered} \qquad (152)$$

$$O(\nu) \qquad \begin{gathered} \dot{u}_1 + \frac{\partial}{\partial x}\left(2u_0 u_1\right) + \frac{\partial}{\partial \zeta}\left(u_0 v_1 + v_0 u_1\right) + \frac{\partial p_1}{\partial x} = \frac{\partial^2 u_1}{\partial \zeta^2} + \frac{\partial^2 u_0}{\partial x^2} \\ \dot{v}_0 + \frac{\partial}{\partial x}\left(u_0 v_0\right) + \frac{\partial}{\partial \zeta}\left(v_0^{\,2}\right) + \frac{\partial p_1}{\partial \zeta} = \frac{\partial^2 v_0}{\partial \zeta^2} \\ \frac{\partial u_1}{\partial x} + \frac{\partial v_1}{\partial \zeta} = 0 \end{gathered} \qquad (153)$$

4.2.1.3 Matching

Once the solutions to the two separate expansions are found, they must be "matched" to one another to deliver the full composite solution across the entire domain. The matching process may be though of as the imposition of boundary conditions in the region of common validity. For the zero-order solutions, the process is fairly simple. The higher order expansions require more effort.

The zero-order outer layer solution simply is the Euler equations. In the zero-order approximation the Euler solution is valid everywhere in the fluid domain except for an infinitesimally thin region around the plate, specifically, a thin neighborhood around y=0. The inner layer expansion is valid in the entire region of the transformed coordinate ζ.

However, since $\zeta = y/\nu^{1/2}$ the limit as ν approaches zero requires that ζ approach ∞ (for finite y). Thus, our zero-order matching conditions are given by

$$\begin{aligned} &\lim_{\zeta\to\infty} u_0(x,\zeta;t) = \lim_{y\to 0} U_0(x,y;t) = U_0(x,0;t), \\ &\lim_{\zeta\to\infty} v_0(x,\zeta;t) = \lim_{y\to 0} \nu^{1/2} V_0(x,y;t) = \nu^{1/2} V_0(x,0;t), \\ &\lim_{\zeta\to\infty} p_0(x,\zeta;t) = \lim_{y\to 0} P_0(x,y;t) = P_0(x,0;t). \end{aligned} \tag{154}$$

Matching of the first-order solution is more involved. The two solutions are required to agree to first order over an open interval of common validity, say $y \in (y_1, y_2)$. To do this typically one picks an intermediate variable of the form $y_\eta = y/\nu^\eta$ and transforms the two opposing solutions to this variable. The solutions are then required to agree to first order for η in (0,1).

4.2.2 Example - The Flat Plate Problem

To demonstrate the use of the above results we compute the steady-state asymptotic solution to our infinitesimally thin flat plate. Referring to Figure 10, the plate is placed edgewise in a uniform flow at the position $y=0$. It is well known that there exists a semi-analytic solution for the zero-order system, the Blasius solution [8]. Here, however, we add the first-order corrections.

4.2.2.1 The Zero-Order Solution

Due to the presence of viscosity, no-slip boundary conditions are imposed upon the fluid at the plate boundary. At the edge of the boundary layer (i.e., the limit as ζ approaches ∞) the fluid must assume the uniform flow conditions. The outer layer solution is simply the solution to the Euler equations, in this case the uniform flow

$$\begin{aligned} U_0 &= u_e(x,y;t) = U_\infty = \text{const} \\ V_0 &= v_e(x,y;t) = 0 \\ P_0 &= p_e(x,y;t) = P_\infty = \text{const} \end{aligned} \tag{155}$$

(As mentioned, the subscript e indicates the solution to the Euler equations.) The zero-order matching conditions require that

$$\begin{aligned} &\lim_{\zeta\to\infty} u_0(x,\zeta;t) = \lim_{y\to 0} u_e(x,y;t) = U_\infty, \\ &\lim_{\zeta\to\infty} v_0(x,\zeta;t) = \lim_{y\to 0} \nu^{1/2} v_e(x,y;t) = 0, \\ &\lim_{\zeta\to\infty} p_0(x,\zeta;t) = \lim_{y\to 0} p_e(x,y;t) = P_\infty. \end{aligned} \tag{156}$$

Along with these matching conditions, we also have the following boundary condition on our asymptotic system:

$$\begin{matrix} u_0(x,y;t) = U_\infty \\ v_0(x,y;t) = 0 \end{matrix} \quad \text{for } x<0 \qquad \begin{matrix} u_0(x,0;t) = 0 \\ v_0(x,0;t) = 0 \end{matrix} \quad \text{for } x>0 \tag{157}$$

In the region $x>0$, the steady state solution to the zero-order boundary layer is given by the Blasius solution [8], which is

$$u_0(x,\zeta) = U_\infty f'(\eta(x,\zeta))$$

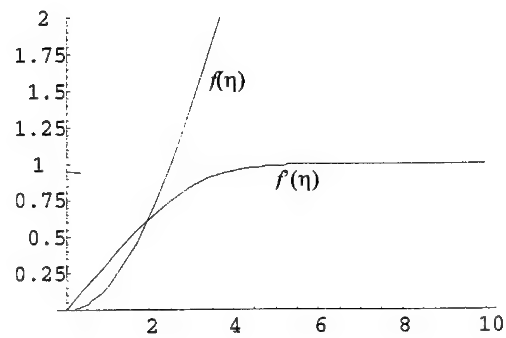
$$v_0(x,\zeta) = \frac{1}{2}\frac{\eta}{\zeta}[\eta f'(\eta) - f(\eta)] \qquad \eta(x,\zeta) \equiv \zeta\left(\frac{U_\infty}{x}\right)^{\frac{1}{2}} \qquad (158)$$

$$p_0(x,\zeta) = P_\infty$$

where the function $f(\eta)$ solves the differential system

$$\frac{d^3 f}{d\eta^3} + \frac{1}{2} f \frac{d^2 f}{d\eta^2} = 0 \qquad \begin{aligned} f(0) &= 0 \\ f'(0) &= 0 \\ \lim_{\eta\to\infty} f'(\eta) &= 1 \end{aligned} \qquad (159)$$

In the region $x<0$ the solution is simply the uniform flow conditions. Thus, the fluid does not "feel" the plate until it reaches the region $x>0$. Physically, we expect a wake extending into the region $x<0$ around the front edge of the plate, the point $(x,y)=(0,0)$. However, our analysis does not account for this and it is well known that the Blasius solution is not valid in a neighborhood of (0,0) – in fact, it is singular.



• Figure 11: The Blasius function f and its first derivative

We list a few properties of the function $f(\eta)$ for later use. It can be shown that the boundary conditions on $f(\eta)$ imply [8]

$$f''(0) = \alpha \approx 0.33206. \qquad (160)$$

We also have the useful facts that

$$\lim_{\eta\to\infty} (\eta - f(\eta)) = \beta \approx 1.7208, \qquad (161)$$

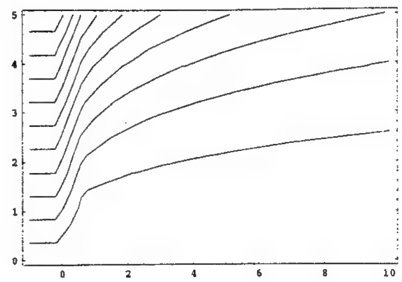
and

$$f''(\eta) \xrightarrow[\eta\to\infty]{} e^{-\eta}. \qquad (162)$$

Finally, the first few terms in the Taylor expansion are

$$f(\eta) = \alpha\frac{\eta^2}{2!} - \frac{\alpha^2}{2}\frac{\eta^5}{5!} + \frac{11\alpha^3}{4}\frac{\eta^8}{8!} - + \ldots \qquad (163)$$

Using the initial values for $f'(\eta)$ and $f''(\eta)$ the function $f(\eta)$ may be easily computed numerically. A plot of $f(\eta)$ and $f'(\eta)$ is shown in Figure 11. Note that $f'(\eta)$ approaches its limiting value around $\eta \approx 5$. The streamlines for the Blasius solution for $U_\infty=1$ are shown in Figure 12. These are simply the level sets of the stream function $\psi=U_\infty f(\eta)$ and, consequently, the level sets of $\eta(x,\zeta)$. Due to the discontinuity in the plate at $x=0$ we can see how the solution is invalid in the neighborhood of $(x,y)=0$. Collecting these results we find that the zero-order composite solution for the flat plate problem is given by



• Figure 12: Streamlines for Blasius solution

$$
\begin{aligned}
&u(x,y)=U_\infty && && u(x,y)=U_\infty f'\left(y\left(\frac{U_\infty}{\nu x}\right)^{\frac{1}{2}}\right) && \\
&v(x,y)=0 && \text{for } x<0 && v(x,y)=\frac{U_\infty}{2}\frac{y}{x}f'\left(y\left(\frac{U_\infty}{\nu x}\right)^{\frac{1}{2}}\right)-\frac{\nu^{1/2}}{2}\left(\frac{U_\infty}{x}\right)^{\frac{1}{2}}f\left(y\left(\frac{U_\infty}{\nu x}\right)^{\frac{1}{2}}\right) && \text{for } x>0 \\
&p(x,y)=P_\infty && && p(x,y)=P_\infty &&
\end{aligned}
\tag{164}
$$

4.2.2.2 The First-Order Solution

Substituting the zero-order outer solution into the first-order outer layer equations yields the system we must solve for U_1, V_1 and P_1. Since the Euler solution is simply the free-flow conditions $U_0=U_\infty$, $V_0=0$, $P_0=P_\infty$, we have

$$
\begin{aligned}
U_\infty\frac{\partial U_1}{\partial x}+\frac{\partial P_1}{\partial x}&=0\\
U_\infty\frac{\partial V_1}{\partial x}+\frac{\partial P_1}{\partial y}&=0\\
\frac{\partial U_1}{\partial x}+\frac{\partial V_1}{\partial y}&=0
\end{aligned}
\tag{165}
$$

These equations can be combined and manipulated to yield the following results:

$$
\begin{aligned}
U_1(x,y)&=-\frac{P_1(x,y)}{U_\infty}\\
\nabla^2 V_1(x,y)&=0\\
\nabla^2 P_1(x,y)&=0
\end{aligned}
\tag{166}
$$

where ∇^2 denotes the Laplacian operator. The solution to the Laplacian equations can be found using separation of variables. After imposing the boundary conditions at $x=0$ and $y\to\infty$ we find

$$V_1(x,y) = \int_0^\infty C_V(k) e^{-ky} \sin kx\, dk$$
$$P_1(x,y) = \int_0^\infty C_P(k) e^{-ky} \sin kx\, dk \tag{167}$$

where $C_p(k)$ and $C_V(k)$ are arbitrary functions of the separation parameter k. Actually, from inspection we see that $C_p(k)$ is the Fourier sine transform of the function $P_1(x,0)$, likewise for $C_V(k)$ and $V_1(x,0)$. Thus, these functions are determined from the matching conditions between the two layers.

Now consider the inner layer. Referring to the first-order inner layer equations (the O(ν) equations), the second equation in that set contains expression for u_0, v_0 and p_1 only; that is, p_1 is the only unknown. By substituting the Blasius solution for u_0 and v_0 into this equation we get the following relation:

$$\frac{\partial p_1}{\partial \zeta} = \frac{1}{4}\left(\frac{U_\infty}{x}\right)^{\frac{3}{2}} \left[\eta (f'(\eta))^2 + 2U_\infty f''(\eta) - U_\infty f(\eta) f'(\eta)\right] \tag{168}$$

It is possible to integrate this expression analytically to obtain an expression for $p_1(x,\zeta)$. First we define the function $g(\eta)$ by

$$g(\eta) \equiv \int_0^\eta [f'(\sigma)]^2\, d\sigma. \tag{169}$$

We can find the expression for $g(\eta)$ by integrating the differential equation for $f(\eta)$. Upon integrating the second term by parts we find

$$g(\eta) = 2f''(\eta) + f(\eta)f'(\eta) - 2\alpha \tag{170}$$

With this relation one may determine that (again using integration by parts)

$$\begin{aligned}\int_0^\eta \sigma [f'(\sigma)]^2 d\sigma &= [\eta g(\eta)]_0^\eta - \int_0^\eta g(\sigma)\, d\sigma \\ &= 2\eta f''(\eta) + \eta f(\eta) f'(\eta) - 2f'(\eta) - \frac{1}{2}[f(\eta)]^2\end{aligned} \tag{171}$$

Thus, the expression for p_1 is given by

$$p_1(x,\zeta) = \frac{1}{4}\frac{U_\infty}{x}\left[2\eta f''(\eta) + \eta f(\eta) f'(\eta) - (f(\eta))^2\right] + C_1(x) \tag{172}$$

where $C_1(x)$ is the constant of integration with respect to ζ. To find $C_1(x)$ it is necessary to apply the boundary condition at x=0, namely $p_1(0,\zeta)$=0.

$$\begin{aligned}\lim_{x\to 0} p_1(x,\zeta) &= \lim_{\eta\to\infty}\frac{1}{2}\frac{\eta^3}{\zeta^2}e^{-\eta} + \lim_{\eta\to\infty}\frac{1}{4}\frac{\eta^2}{\zeta^2}f(\eta)[\eta f'(\eta) - f(\eta)] + C_1\left(U_\infty\zeta^2/\eta^2\right) \\ &= \lim_{\eta\to\infty}\frac{1}{4}\frac{\eta^2}{\zeta^2}f(\eta)[\eta - f(\eta)] + C_1\left(U_\infty\zeta^2/\eta^2\right) \\ &= \lim_{\eta\to\infty}\frac{1}{4}\frac{\eta^2}{\zeta^2}\beta f(\eta) + C_1\left(U_\infty\zeta^2/\eta^2\right) \\ &= 0\end{aligned} \tag{173}$$

This implies that $C_1(x)$ is of the form

$$C_1(x) = -\frac{\beta}{4}\frac{U_\infty}{x} f\left(\sqrt{U_\infty / x}\right) + O(x^\gamma) \tag{174}$$

where $\gamma>0$. With such an expression for $C_1(x)$ the pressure is zero at x=0, except at the point $(x,\zeta)=(0,0)$ where the Blasius solution is known to be singular.

Next we attempt to match the inner and outer solutions for the pressure. To do this we introduce the variable

$$\bar{y} \equiv \frac{y}{\nu^\gamma} \tag{175}$$

where $\gamma\in(0,1/2)$. Converting both the full (i.e., up to order $O(\nu)$) inner and outer expressions to this variable, they should agree to first order for γ in the interval (0,1/2) as ν approaches zero. Letting

$$\xi \equiv \frac{\bar{y}}{\nu^{1/2-\gamma}}\left(\frac{U_\infty}{x}\right)^{\frac{1}{2}} \tag{176}$$

we get

$$\begin{aligned}p_1(x,\bar{y}) &= P_\infty + \nu\frac{1}{4}\frac{U_\infty}{x}\left[2\xi f''(\xi) + \xi f(\xi)f'(\xi) - \left(f(\xi)\right)^2\right] + \nu C_1(x) \\ &\approx P_\infty + \nu\frac{1}{4}\frac{U_\infty}{x}\beta f(\xi) + \nu C_1(x) \\ &\approx P_\infty + \nu^{\gamma+1/2}\frac{1}{4}\left(\frac{U_\infty}{x}\right)^{\frac{3}{2}}\beta^2\bar{y} + \nu\left[C_1(x) - \frac{1}{4}\frac{U_\infty}{x}\beta\right]\end{aligned} \tag{177}$$

and

$$\begin{aligned}P_1(x,y) &= P_\infty + \nu\int_0^\infty C_P(k)\,e^{-k\nu^\gamma\bar{y}}\sin kx\,dk \\ &\approx P_\infty + \nu\int_0^\infty C_P(k)\,\sin kx\,dk - \nu^{1+\gamma}\bar{y}\int_0^\infty C_P(k)\,k\sin kx\,dk\end{aligned} \tag{178}$$

Because of the $O(\nu^{1/2+\gamma})$ term in the inner layer solution, it is impossible to match the solutions to first order. This fact suggests that our original assumption that the solutions could be expanded in powers of ν was probably wrong. A more likely expansion would

then seem to be in powers of $\nu^{1/2}$. However, due to time constraints in Phase I we have not been able to further pursue this avenue of investigation.

4.2.3 Wall Curvature

Here we briefly discuss the situation were the wall is curved, rather than a simple flat plate. Assume that we have a coordinate system as in Figure 9, where the tangential coordinate is given by x. We denote the curvature of the wall by $\kappa(x)$, where $\kappa(x)$ is assumed to be continuous and have a first derivative. Using a boundary layer analysis where ζ is the stretched normal coordinate as before, it can be shown that the curvature will affect the pressure distribution according to the following [8]:

$$\frac{\partial p'}{\partial \zeta} = \nu^{1/2}\kappa(x)u'^2 \qquad (179)$$

In our present expansion of p' this equation would probably have to be added into the $O(1)$ system and would affect the boundary layer directly. However, if we tried an expansion in powers of $\nu^{1/2}$ the curvature would enter into the $O(\nu^{1/2})$ system. From the previous analysis we expect that system to be linear in the field variables. Thus, it may be possible to incorporate wall curvature and achieve some sort of analytic predictions. This condition would be extremely useful if one were to employ the curvature as the control, for example, using MEMS type devices mounted on the body surface. As with the case above, our funding for Phase I ran out and we were not able to pursue this idea as of yet.

4.2.4 Multiple Time Scales

The time derivative of v_0 appears in the first-order equations and not the zero-order equations. However, v_0 and u_0 are not independent, they are related by the divergence conditions. Thus, the time behavior of v_0 will be dictated by the time behavior of u_0 through that condition. Yet it is interesting to investigate the possibility of two time scales, a slow time scale t_1 and a fast time scale t_2. Such a condition would be necessary to introduce the time derivative of v_0 into the zero-order system, which are the boundary layer equations. The form of Eqs. (144) suggests the following definitions

$$t_1 \equiv t, \qquad \text{and} \qquad t_2 \equiv \frac{t}{\nu}, \qquad (180)$$

so that

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial t_1} + \frac{1}{\nu}\frac{\partial}{\partial t_2}, \qquad (181)$$

and the field variables are now functions of x, y, t_1, and t_2 (e.g., $u'(x,y;t_1,t_2)$). Applying this transform to Eqs. (144) yield the system

$$
\begin{gathered}
\frac{\partial u'}{\partial t_1}+\frac{1}{\nu}\frac{\partial u'}{\partial t_2}+u'\frac{\partial u'}{\partial x}+v'\frac{\partial u'}{\partial \zeta}+\frac{\partial p'}{\partial x}=\nu\frac{\partial^2 u'}{\partial x^2}+\frac{\partial^2 u'}{\partial \zeta^2}\\
\nu\frac{\partial v'}{\partial t_1}+\frac{\partial v'}{\partial t_2}+\nu\left(u'\frac{\partial v'}{\partial x}+v'\frac{\partial v'}{\partial \zeta}\right)+\frac{\partial p'}{\partial \zeta}=\nu^2\frac{\partial^2 v'}{\partial x^2}+\nu\frac{\partial^2 v'}{\partial \zeta^2}\\
\frac{\partial u'}{\partial x}+\frac{\partial v'}{\partial \zeta}=0
\end{gathered}
\tag{182}
$$

Inserting the expansion of Eqs. (145) into the above yields the following systems of asymptotic expressions

$O(1/\nu)$
$$
\frac{\partial u_0}{\partial t_2}=0 \tag{183}
$$

$O(1)$
$$
\begin{gathered}
\frac{\partial u_1}{\partial t_2}+\frac{\partial u_0}{\partial t_1}+u_0\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_0}{\partial x}=\frac{\partial^2 u_0}{\partial \zeta^2}\\
\frac{\partial v_0}{\partial t_2}+\frac{\partial p_0}{\partial \zeta}=0\\
\frac{\partial u_0}{\partial x}+\frac{\partial v_0}{\partial \zeta}=0
\end{gathered}
\tag{184}
$$

$O(\nu)$
$$
\begin{gathered}
\frac{\partial u_1}{\partial t_1}+u_0\frac{\partial u_1}{\partial x}+u_1\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_1}{\partial \zeta}+v_1\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_1}{\partial x}=\frac{\partial^2 u_1}{\partial \zeta^2}+\frac{\partial^2 u_0}{\partial x^2}\\
\frac{\partial v_1}{\partial t_2}+\frac{\partial v_0}{\partial t_1}+u_0\frac{\partial v_0}{\partial x}+v_0\frac{\partial v_0}{\partial \zeta}+\frac{\partial p_1}{\partial \zeta}=\frac{\partial^2 v_0}{\partial \zeta^2}\\
\frac{\partial u_1}{\partial x}+\frac{\partial v_1}{\partial \zeta}=0
\end{gathered}
\tag{185}
$$

There also exist boundary conditions associated with the above system. This situation is more complicated than the single time case since now we have coupling between u_0 and u_1 in the zero-order ($O(1)$) equations. However, it is possible to decouple the system by employing the Fredholm alternative theorem [7].

Note that the initial conditions in Eq. (184) require that $u_1(x,y;t_1,t_2)=0$ when $t=t_1=t_2=0$. We now impose the additional constraint that $u_1=0$ at $t_2=T/\nu$, where we shall take $t=T$ as the final time. Thus,

$$
u_1(x,y;t_1,\cdot)\in X\equiv\left\{h\in C^1([0,T/\nu))\,\middle|\,h(0)=0,\,h(T/\nu)=0\right\}. \tag{186}
$$

For clarity let L denote the operator $\partial/\partial t_2$ and define the function space Y according to

$$
Y\equiv\left\{h\in C^0([0,T/\nu))\right\} \tag{187}
$$

so that

$$
\begin{aligned}
L:X&\to Y\\
h&\mapsto \partial h/\partial t_2
\end{aligned}
\tag{188}
$$

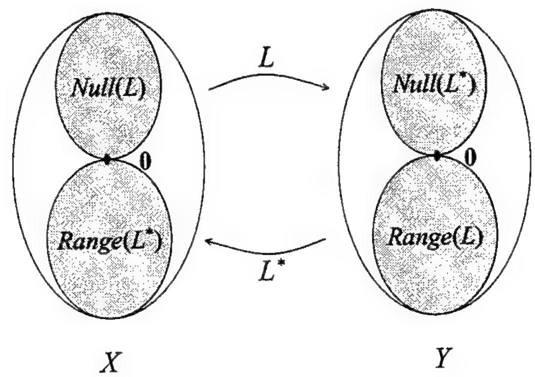
Both X and Y can be given a Hilbert space structure by introducing the inner product

$$\langle g,h\rangle \equiv \int_0^{\frac{T}{v}} g(t_2)h(t_2)\,dt_2 \qquad (189)$$

Then the adjoint operator L^* is easily found to be

$$\begin{aligned} L^*&:X\to Y \\ h&\mapsto -\partial h/\partial t_2 \end{aligned} \qquad (190)$$

using a simple integration by parts. From the lack of boundary conditions in the space Y we see that L^* has a nontrivial null space. To insure that the first of Eqs. (184) has a solution, the Fredholm alternative requires that the terms involving the zero-order components must be in the range of $L=\partial/\partial t_2$. Recall from Hilbert space theory that the range of the operator L is always orthogonal to the null space of the adjoint L^*. That is,



• Figure 13: Schematic of Fredholm alternative

$$Y \supset Range(L) \perp Null(L^*) \subset Y. \qquad (191)$$

This condition is represented graphically in Figure 13 where the only intersection between $Range(L)$ and $Null(L^*)$ is the zero vector (likewise for $Range(L^*)$ and $Null(L)$). Since the null space of $L^*=\partial/\partial t_2$ is the set of constant functions on $[0, T/v)$, the orthogonality relation requires that

$$\int_0^{T/v}\left(\frac{\partial u_0}{\partial t_1}+u_0\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_0}{\partial x}-\frac{\partial^2 u_0}{\partial \zeta^2}\right)dt_2=0 \qquad (192)$$

for all x, y, and t_1. Practically, this condition is met only when the integrand is identically zero (another alternative is that the integrand is periodic in t_2, an unlikely event). Therefore, $\partial u_1/\partial t_2=0$ so that u_1 is also independent of the fast time scale. Applying a similar analysis to the second of Eqs. (184) implies that $\partial v_1/\partial t_2=0$. Collecting these results we have

Temporal boundary conditions:

$$
\begin{array}{lll}
u_0(x,y;0,0)=u_i(x,y) & v_0(x,y;0,0)=v_i(x,y) & p_0(x,y;0,0)=p_i(x,y) \\
u_1(x,y;0,0)=0 & v_1(x,y;0,0)=0 & p_1(x,y;0,0)=0
\end{array}
\tag{193}
$$

$$
u_1(x,y;t,T/\nu)=0 \qquad v_1(x,y;t,T/\nu)=0
$$

Matching conditions:

$$
\begin{aligned}
&\lim_{\zeta\to\infty} u_0(x,\zeta;t)=\lim_{y\to 0} u_e(x,y;t), \\
&\lim_{\zeta\to\infty} v_0(x,\zeta;t)=\lim_{y\to 0} \nu^{1/2} v_e(x,y;t), \\
&\lim_{\zeta\to\infty} p_0(x,\zeta;t)=\lim_{y\to 0} p_e(x,y;t).
\end{aligned}
\tag{194}
$$

Independence of fast time scale:

$$
\frac{\partial u_0}{\partial t_2}=0 \quad \frac{\partial u_1}{\partial t_2}=0 \quad \frac{\partial v_1}{\partial t_2}=0
\tag{195}
$$

Dynamics equations $O(1)$:

$$
\begin{aligned}
\frac{\partial u_0}{\partial t_1}+u_0\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_0}{\partial x}&=\frac{\partial^2 u_0}{\partial \zeta^2} \\
\frac{\partial v_0}{\partial t_2}+\frac{\partial p_0}{\partial \zeta}&=0 \\
\frac{\partial u_0}{\partial x}+\frac{\partial v_0}{\partial \zeta}&=0
\end{aligned}
\tag{196}
$$

Dynamics Equations $O(\nu)$:

$$
\begin{aligned}
\frac{\partial u_1}{\partial t_1}+u_0\frac{\partial u_1}{\partial x}+u_1\frac{\partial u_0}{\partial x}+v_0\frac{\partial u_1}{\partial \zeta}+v_1\frac{\partial u_0}{\partial \zeta}+\frac{\partial p_1}{\partial x}&=\frac{\partial^2 u_1}{\partial \zeta^2}+\frac{\partial^2 u_0}{\partial x^2} \\
\frac{\partial v_0}{\partial t_1}+u_0\frac{\partial v_0}{\partial x}+v_0\frac{\partial v_0}{\partial \zeta}+\frac{\partial p_1}{\partial \zeta}&=\frac{\partial^2 v_0}{\partial \zeta^2} \\
\frac{\partial u_1}{\partial x}+\frac{\partial v_1}{\partial \zeta}&=0
\end{aligned}
\tag{197}
$$

These equations could provide useful when attempting to design a controller where the response time is comparable to the flow rate reaction. This possibility was mentioned when discussing the control of the discrete case.

4.3 Conclusion

As we have mentioned we are still investigating the control aspects of this project and our approach so far has been rather pragmatic. It is hoped that the foray into the theoretical structure of the Navier-Stokes equations can provide some practical insight into fluid-flow controller design. This is especially true with respect to the last sections where we hope to determine the dynamics of active boundary layer stabilization.

5 REFERENCES

[1] J.E. Akin, *Finite Elements for Analysis and Design* (Academic Press, London, 1994).

[2] K.J. Bathe, *Finite Element Procedures in Engineering Analysis* (Prentice-Hall, 1982).

[3] S.C. Brenner and L.R. Scott, *The Mathematical Theory of Finite Element Methods* (Springer-Verlag, 1994, New York).

[4] R. Glowinski, T.W. Pan, and J. Periaux, "A fictitious domain method for Dirichlet problem and applications", *Comput. Methods. Appl. Mech. Eng.*, Vol. 111 (1994), pp. 283-303.

[5] D. Peaceman and M. Rachford, "The numerical solution of parabolic and elliptic differential equations", *J. SIAM*, Vol. 3 (1955), pp. 28-41.

[6] L. Prandtl and O. G. Tietjens, Applied Hydro- and Aeromechanics. New York: McGraw-Hill Book Company, 1934.

[7] I. Stakgold, *Green's Functions and Boundary Value Problems* (Wiley, New York, 1979), pp. 207-214, 337-338.

[8] R.E. Meyer, *Introduction to Mathematical Fluid Dynamics* (Dover, New York, 1971), Chapt. 4.

[9] M.H. Holmes, *Introduction to Perturbation Methods* (Springer-Verlag, New York, 1995), Chapt. 2.

[10] J. Zabczyk, *Mathematical Control Theory: An Introduction* (Birkhauser, Boston, 1992), Part IV.

[11] M. Renardy and R.C. Rogers, *An Introduction to Partial Differential Equations* (Springer-Verlag, New York, 1993), Chapt. 12.

[12] R. Glowinski, "Splitting methods for the numerical solution of the incompressible Navier-Stokes equations", *Vistas in Applied Mathematics*, A.V. Balakrishnan, A.A. Doronitsyn, and J.L. Lions Eds. (Optimization Software, New York, 1986), pp. 57-95.

[13] R. Glowinski, "Finite element methods for the numerical simulation of incompressible viscous flow. Introduction to the control of the Navier-Stokes equations", *Lectures in Applied Mathematics*, Vol. 28, AMS, Providence, R.I. (1991), pp. 219-301.

[14] R. Glowinski, T.W. Pan, and J. Periaux, "A fictitious domain method for external incompressible viscous flow modeled by Navier-Stokes equation", *Comput. Methods. Appl. Mech. Eng.*, Vol. 112 (1994), pp. 133-148.

[15] R. Glowinski, A.J. Kearsley, and J. Periaux, "Numerical simulation and optimal shape for viscous flow by a fictitious domain method", *Internat. J. Numer. Methods in Fluids*, Vol. 20 (1995), pp. 695-711.

[16] R. Glowinski, *Numerical Methods for Nonlinear Variational Problems* (Springer-Verlag, New York, 1984).

[17] R. Glowinski and P. Le Tallec, *Augmented Lagrangian and Operator-Splitting Methods in Nonlinear Mechanics* (SIAM, Philadelphia, 1989).

[18] M.D. Gunzberger, *Finite Element Methods for Incompressible Flows: A Guide to Theory, Practice, and Algorithms* (Academic Press, Boston, 1989).

[19] D. Potter, *Computational Physics* (Wiley, Chichester, 1973).

[20] J.F. Wendt (Ed.), *Computational Fluid Dynamics* (Springer-Verlag, Berlin, 1996).

[21] C.R. Doering and J.D. Gibbon, *Applied Analysis of the Navier-Stokes Equations* (Cambridge University Press, 1995).

[22] R. Peyret and T.D. Taylor, *Computational Methods for Fluid Flow* (Springer-Verlag, New York, 1983).

[23] C.A.J. Fletcher, *Computational Techniques for Fluid Dynamics, Second Ed.*, Volumes I and II (Springer-Verlag, Berlin, 1991).

APPENDIX A: Source Listing

The Mathematica source file used in the boundary layer analysis is listed here. The source symbolically computes the two-dimensional incompressible Navier-Stokes equations for any coordinate system. The general coordinates are (u1,u2) and their mapping into cartesian space is supplied by the functions x1(u1,u2) and x2(u1,u2). The principles of differential geometry are then applied to generation the form of the Navier-Stokes equations in the general coordinates (u1,u2). The current listing is for stretched polar coordinates using a stretching parameter α. Specifically, the radial coordinate is stretched around the value u1=1 by a factor $1/\alpha$. At the end of the file the stretching parameter α is taken to be v^2 and limits are taken for the regime $\alpha \to 0$. The boundary layer equations are then recovered.

Fluid Flow in Polar Coordinates

■ Preliminaries

```
ClearAll[]
```

■ Load Packages

```
<< "G:\Wolfram Research\Mathematica3.0\AddOns\StandardPackages\Graphics\PlotField.m"

Names["Graphics'PlotField'*"]

{}
```

■ Our favorite standard basis

```
e1 = {1, 0}

{1, 0}

e2 = {0, 1}

{0, 1}
```

■ Stretched Polar Coordinates

We expand the traditional polar coordinate r about r=1 by the factor α.
Our new coordinate being denoted by ζ, the result is $\zeta = (r-1)/\alpha$. OR

$$r = 1 + \alpha\zeta$$

The Cartesian map is then given by

$$x(\zeta,\theta) = x_1(\zeta,\theta)\,\mathbf{e_1} + x_2(\zeta,\theta)\,\mathbf{e_2}$$
$$= (1+\alpha\zeta)\cos\theta\,\mathbf{e_1} + (1+\alpha\zeta)\sin\theta\,\mathbf{e_2}$$

■ The general coordinates - (u1,u2)

```
Clear[u1, u2]

u = {u1, u2}

{u1, u2}
```

- **Set up the Cartesian map x[u1,u2] = {x1[u1,u2], x2[u1,u2]}**

```
x1[u1_, u2_] = (1 + α u1) Cos[u2]
(1 + u1 α) Cos[u2]

x2[u1_, u2_] = (1 + α u1) Sin[u2]
(1 + u1 α) Sin[u2]

x[u1_, u2_] = x1[u1, u2] e1 + x2[u1, u2] e2
{(1 + u1 α) Cos[u2], (1 + u1 α) Sin[u2]}
```

- **Set up the tangent plane basis vectors: $T_{(u1,u2)}R$ = span{g1,g2}**

```
g1[u1_, u2_] = ∂u1 x[u1, u2]
{α Cos[u2], α Sin[u2]}

g2[u1_, u2_] = ∂u2 x[u1, u2]
{-(1 + u1 α) Sin[u2], (1 + u1 α) Cos[u2]}
```

- **Now compute the Jacobian and the dual basis for the Cotangent bundle (gr1 and gr2)**

```
G[u1_, u2_] = Transpose[{g1[u1, u2], g2[u1, u2]}]
{{α Cos[u2], -(1 + u1 α) Sin[u2]}, {α Sin[u2], (1 + u1 α) Cos[u2]}}

Ginv[u1_, u2_] = Simplify[Inverse[G[u1, u2]]]
```

$$\left\{\left\{\frac{\text{Cos}[u2]}{\alpha}, \frac{\text{Sin}[u2]}{\alpha}\right\}, \left\{-\frac{\text{Sin}[u2]}{1+u1\,\alpha}, \frac{\text{Cos}[u2]}{1+u1\,\alpha}\right\}\right\}$$

```
MatrixForm[G[u1, u2]]
```

$$\begin{pmatrix} \alpha\,\text{Cos}[u2] & -(1+u1\,\alpha)\,\text{Sin}[u2] \\ \alpha\,\text{Sin}[u2] & (1+u1\,\alpha)\,\text{Cos}[u2] \end{pmatrix}$$

```
MatrixForm[Ginv[u1, u2]]
```

$$\begin{pmatrix} \frac{\text{Cos}[u2]}{\alpha} & \frac{\text{Sin}[u2]}{\alpha} \\ -\frac{\text{Sin}[u2]}{1+u1\,\alpha} & \frac{\text{Cos}[u2]}{1+u1\,\alpha} \end{pmatrix}$$

```
J[u1_, u2_] = Simplify[Det[G[u1, u2]]]
α (1 + u1 α)

gr1[u1_, u2_] = Ginv[u1, u2][[1]]
```

$$\left\{\frac{\text{Cos}[u2]}{\alpha}, \frac{\text{Sin}[u2]}{\alpha}\right\}$$

```
gr2[u1_, u2_] = Ginv[u1, u2][[2]]
```

$\{-\frac{\mathrm{Sin}[u2]}{1+u1\,\alpha}, \frac{\mathrm{Cos}[u2]}{1+u1\,\alpha}\}$

■ The metric induced from E^2 – denoted g

$$g[u1\_, u2\_] = \mathrm{Simplify}[\begin{pmatrix} g1[u1, u2] . g1[u1, u2] & g1[u1, u2] . g2[u1, u2] \\ g2[u1, u2] . g1[u1, u2] & g2[u1, u2] . g2[u1, u2] \end{pmatrix}]$$

$\{\{\alpha^2, 0\}, \{0, (1+u1\,\alpha)^2\}\}$

```
MatrixForm[g[u1, u2]]
```

$$\begin{pmatrix} \alpha^2 & 0 \\ 0 & (1+u1\,\alpha)^2 \end{pmatrix}$$

■ The contravariant metric (metric for covectors in basis gr1,gr2)

```
ginv[u1_, u2_] = Inverse[g[u1, u2]]
```

General::spell1 :
Possible spelling error: new symbol name "ginv" is similar to existing symbol "Ginv".

$$\{\{\frac{(1+u1\,\alpha)^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\,\alpha^4}, 0\}, \{0, \frac{\alpha^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\,\alpha^4}\}\}$$

```
MatrixForm[ginv[u1, u2]]
```

$$\begin{pmatrix} \frac{(1+u1\,\alpha)^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\,\alpha^4} & 0 \\ 0 & \frac{\alpha^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\,\alpha^4} \end{pmatrix}$$

■ Christoffel Symbols

We set up the symbols so that $\Gamma^k_{ij}[u1, u2] = \Gamma[u1, u2][[i, j, k]]$

```
Γ1[u1_, u2_] = Transpose[
               Simplify[
               Ginv[u1, u2] . D[G[u1, u2], u1]]]
```

$\{\{0, 0\}, \{0, \frac{\alpha}{1+u1\,\alpha}\}\}$

```
Γ2[u1_, u2_] = Transpose[
               Simplify[
               Ginv[u1, u2] . D[G[u1, u2], u2]]]
```

$\{\{0, \frac{\alpha}{1+u1\,\alpha}\}, \{-\frac{1+u1\,\alpha}{\alpha}, 0\}\}$

```
Γ[u1_, u2_] = Join[{Γ1[u1, u2]}, {Γ2[u1, u2]}]
```

$\{\{\{0, 0\}, \{0, \frac{\alpha}{1+u1\,\alpha}\}\}, \{\{0, \frac{\alpha}{1+u1\,\alpha}\}, \{-\frac{1+u1\,\alpha}{\alpha}, 0\}\}\}$

■ Check them

```
Γ[u1, u2][[1, 2, 2]]
```

$$\frac{\alpha}{1 + u1\,\alpha}$$

```
Γ[u1, u2][[2, 1, 2]]
```

$$\frac{\alpha}{1 + u1\,\alpha}$$

```
Γ[u1, u2][[2, 2, 1]]
```

$$-\frac{1 + u1\,\alpha}{\alpha}$$

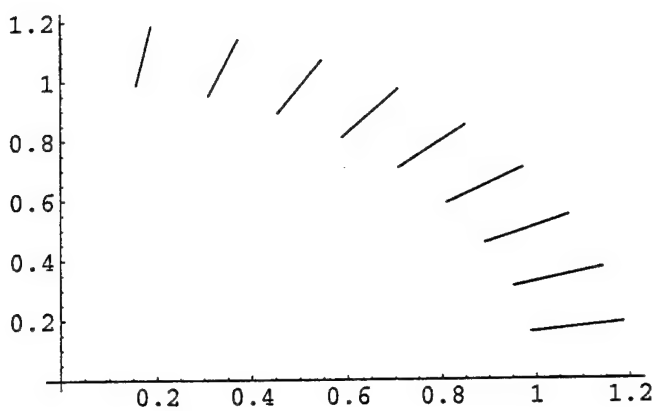
■ View the Constant Coördinate Lines in R2

We'll do this for α=0.2 and (u1,u2)∈[0,1]×[0,π/2]

```
α = 0.2
```

0.2

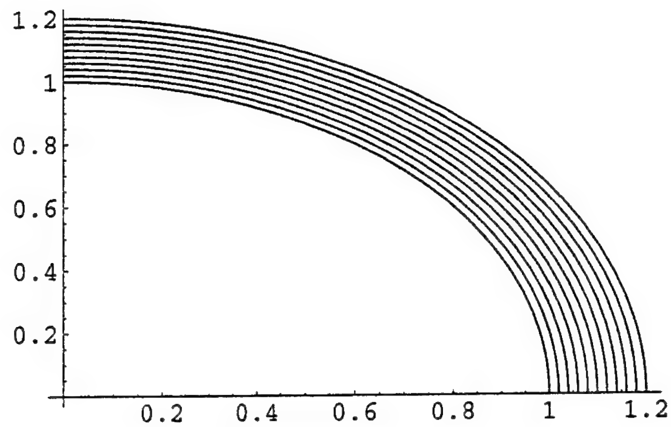
```
tblLines1[u1_] = Table[x[u1, i 0.1 Pi / 2], {i, 0, 10}];
ParametricPlot[Evaluate[tblLines1[u1]], {u1, 0, 1}]
```



- Graphics -

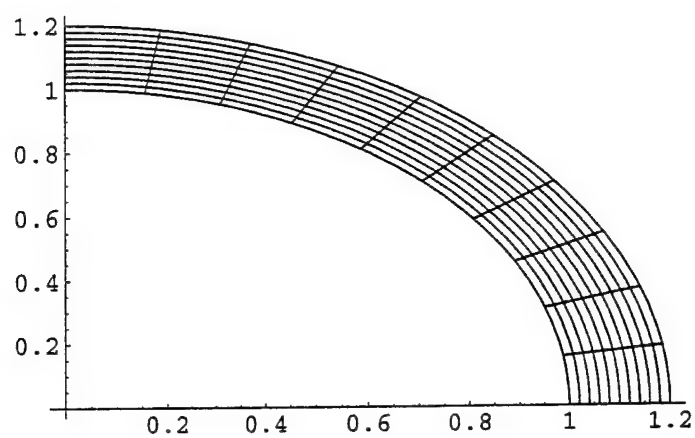
```
tblLines2[u2_] = Table[x[i 0.1, u2 Pi / 2], {i, 0, 10}];
```

```
ParametricPlot[Evaluate[tblLines2[u2]], {u2, 0, 1}]
```



- Graphics -

```
tblLines[x_] = Join[tblLines1[x], tblLines2[x]];
ParametricPlot[Evaluate[tblLines[x]], {x, 0, 1}]
```



- Graphics -

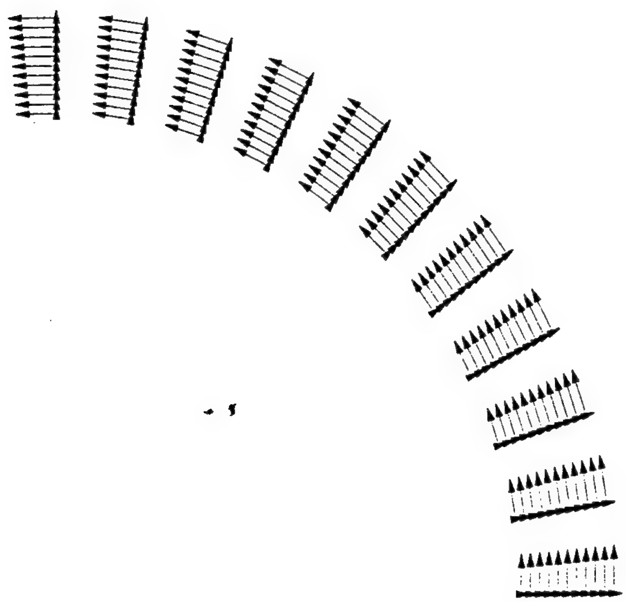
■ View the Polar Tangent Bundle

```
tblTang1 = Table[{x[0.1 i, Pi / 20 j], g1[0.1 i, Pi / 20 j]},
                {i, 0, 10}, {j, 0, 10}];

tblTang2 = Table[{x[0.1 i, Pi / 20 j], g2[0.1 i, Pi / 20 j]},
                {i, 0, 10}, {j, 0, 10}];
```

```
tblTang = Flatten[Join[tblTang1, tblTang2], 1];
```

```
ListPlotVectorField[tblTang, VectorHeads -> True, ScaleFactor -> 0.1]
```



```
- Graphics -
```

- **Now we have to clear off α for symbolic computation**

```
g2[u1, u2]
```

```
{-(1 + 0.2 u1) Sin[u2], (1 + 0.2 u1) Cos[u2]}
```

```
Clear[α]
```

```
g2[u1, u2]
```

```
{-(1 + u1 α) Sin[u2], (1 + u1 α) Cos[u2]}
```

■ Conversion Between Covectors, Vectors, and Physical Components

■ Covectors to Vectors

```
Sub2Sup[v : {_, _}] :=
  Simplify[
   {Sum[ginv[u1, u2][[1, i]] v[[i]], {i, 2}],
    Sum[ginv[u1, u2][[2, i]] v[[i]], {i, 2}]}
   ]
```

Vectors to Covectors

```
Sup2Sub[v : {_, _}] :=
  Simplify[
   {Sum[g[u1, u2][[1, i]] v[[i]], {i, 2}],
    Sum[g[u1, u2][[2, i]] v[[i]], {i, 2}]}
   ]
```

Vectors to Physical Components

```
Sup2Phys[v : {_, _}] :=
 Simplify[
  {v[[1]] / √ginv[u1, u2][[1, 1]] ,
   v[[2]] / √ginv[u1, u2][[2, 2]] }
 ]
```

Covectors to Physical Components

```
Sub2Phys[v : {_, _}] :=
 Simplify[
  {v[[1]] / √g[u1, u2][[1, 1]] ,
   v[[2]] / √g[u1, u2][[2, 2]] }
 ]
```

```
General::spell1 :
 Possible spelling error: new symbol name "Sub2Phys" is similar to existing symbol "Sup2Phys".
```

The Covariant Derivative

The Covariant Derivative of (Contravariant) Vectors

dCovSup[i, v] = $\{\nabla_i v^1, \nabla_i v^2\}$

```
dCovSup[i_, v : {_, _}] :=
   {D[v[[1]], u[[i]]], D[v[[2]], u[[i]]]}
 + {Sum[Γ[u1, u2][[i, j, 1]] v[[j]], {j, 2}],
    Sum[Γ[u1, u2][[i, j, 2]] v[[j]], {j, 2}]}
```

The Covariant Derivative of Covariant Vectors (Covectors)

dCovSub[i, v] = $\{\nabla_i v_1, \nabla_i v_2\}$

```
dCovSub[i_, v : {_, _}] :=
  {D[v[[1]], u[[i]]], D[v[[2]], u[[i]]]}
 - {Sum[Γ[u1, u2][[i, 1, k]] v[[k]], {k, 2}],
   Sum[Γ[u1, u2][[i, 2, k]] v[[k]], {k, 2}]}
```

```
General::spell1 :
 Possible spelling error: new symbol name "dCovSub" is similar to existing symbol "dCovSup".
```

■ Define Various Calculus Operators

■ These operators take scalars to covectors

```
Grad[f_] := {D[f, u[[1]]], D[f, u[[2]]]}
```

■ These operators take scalars to vectors

```
Curl2Vector[f_] :=
 Simplify[
  (1 / J[u1, u2]) {D[f, u2] , -D[f, u1]}
  ]
```

■ These operators take covectors to scalars

```
Curl2Scalar[v : {_, _}] :=
 Simplify[
  (1 / J[u1, u2])
   (dCovSub[1, v] . e2 - dCovSub[2, v] . e1)
  ]
```

■ These operators take vectors to scalars

```
Diverg[v : {_, _}] := dCovSup[1, v] . e1
                     + dCovSup[2, v] . e2
```

■ These operators take scalars to scalars

```
LaplacianScalar[f_] := Sum[ ginv[u1, u2][[i]] . dCovSub[i, Grad[f]], {i, 2} ]
```

■ These operators take vectors to vectors

```
GradSup[f_] := Module[ {df}, df = Grad[f];
  Simplify[
   {Sum[ ginv[u1, u2][[i, 1]] df[[i]], {i, 2}],
    Sum[ginv[u1, u2][[i, 2]] df[[i]], {i, 2}]}
   ]
 ]
```

```
'Advec[v : {_, _}] := v[[1]] dCovSup[1, v]
                     + v[[2]] dCovSup[2, v]

LaplacianVector[v : {_, _}] :=
 Simplify[
  {Sum[ginv[u1, u2][[j, k]]
     (dCovSup[j,
      dCovSup[k, v]
        ] . e1), {j, 2}, {k, 2}],
   Sum[ginv[u1, u2][[j, k]]
     (dCovSup[j,
      dCovSup[k, v]
        ] . e2), {j, 2}, {k, 2}]
    }
   ]
```

■ Compute the Vector Calculus Operations in Polar Coordinates

```
Grad[f[u1, u2]]
```

$\{f^{(1,0)}[u1, u2], f^{(0,1)}[u1, u2]\}$

```
GradSup[f[u1, u2]]
```

$\left\{\frac{f^{(1,0)}[u1, u2]}{\alpha^2}, \frac{f^{(0,1)}[u1, u2]}{(1 + u1\,\alpha)^2}\right\}$

```
Diverg[{v1[u1, u2], v2[u1, u2]}]
```

$\frac{\alpha\, v1[u1, u2]}{1 + u1\,\alpha} + v2^{(0,1)}[u1, u2] + v1^{(1,0)}[u1, u2]$

```
Curl2Scalar[{v1[u1, u2], v2[u1, u2]}]
```

$\frac{-v1^{(0,1)}[u1, u2] + v2^{(1,0)}[u1, u2]}{\alpha + u1\,\alpha^2}$

```
Curl2Vector[f[u1, u2]]
```

$\left\{\frac{f^{(0,1)}[u1, u2]}{\alpha + u1\,\alpha^2}, -\frac{f^{(1,0)}[u1, u2]}{\alpha\,(1 + u1\,\alpha)}\right\}$

```
Advec[{v1[u1, u2], v2[u1, u2]}]
```

$$\left\{v2[u1, u2]\left(-\frac{(1 + u1\,\alpha)\, v2[u1, u2]}{\alpha} + v1^{(0,1)}[u1, u2]\right) + v1[u1, u2]\, v1^{(1,0)}[u1, u2],\right.$$
$$\left. v2[u1, u2]\left(\frac{\alpha\, v1[u1, u2]}{1 + u1\,\alpha} + v2^{(0,1)}[u1, u2]\right) + v1[u1, u2]\left(\frac{\alpha\, v2[u1, u2]}{1 + u1\,\alpha} + v2^{(1,0)}[u1, u2]\right)\right\}$$

```
LaplacianScalar[f[u1, u2]]
```

$$\frac{\alpha^2\left(f^{(0,2)}[u1, u2] + \frac{(1+u1\,\alpha)\, f^{(1,0)}[u1,u2]}{\alpha}\right)}{\alpha^2 + 2\, u1\, \alpha^3 + u1^2\, \alpha^4} + \frac{(1 + u1\,\alpha)^2\, f^{(2,0)}[u1, u2]}{\alpha^2 + 2\, u1\, \alpha^3 + u1^2\, \alpha^4}$$

```
LaplacianVector[{v1[u1, u2], v2[u1, u2]}]
```

$$\left\{\frac{1}{\alpha^2 (1+u1\,\alpha)^2}\left(-\alpha^2 v1[u1, u2] - 2\alpha(1+u1\,\alpha)\, v2^{(0,1)}[u1, u2] + \alpha^2 v1^{(0,2)}[u1, u2] + v1^{(2,0)}[u1, u2] + 2\,u1\,\alpha\, v1^{(2,0)}[u1, u2] + u1^2 \alpha^2 v1^{(2,0)}[u1, u2]\right),\right.$$
$$\left.\frac{1}{\alpha^2 (1+u1\,\alpha)^3}\left(-\alpha^2 (1+u1\,\alpha)\, v2[u1, u2] + 2\alpha^3 v1^{(0,1)}[u1, u2] + (1+u1\,\alpha)\left(\alpha^2 v2^{(0,2)}[u1, u2] + 2\alpha\, v2^{(1,0)}[u1, u2] + 2\,u1\,\alpha^2 v2^{(1,0)}[u1, u2] + v2^{(2,0)}[u1, u2] + 2\,u1\,\alpha\, v2^{(2,0)}[u1, u2] + u1^2\alpha^2 v2^{(2,0)}[u1, u2]\right)\right)\right\}$$

Finally: Fluid Flow

Flow around the cylinder

■ First Find Boundary Conditions for Uniform Flow

■ Uniform flow in x-direction in Cartesian

```
VinfCart = {1, 0}
```

```
{1, 0}
```

■ Convert to Polar

```
Vinf[u1_, u2_] = Ginv[u1, u2] . VinfCart
```

$$\left\{\frac{\mathrm{Cos}[u2]}{\alpha}, -\frac{\mathrm{Sin}[u2]}{1+u1\,\alpha}\right\}$$

■ Find Physical Components

```
VinfPhys[u1_, u2_] = {Vinf[u1, u2][[1]] / √ginv[u1, u2][[1, 1]] ,
    Vinf[u1, u2][[2]] / √ginv[u1, u2][[2, 2]] }
```

$$\left\{\frac{\mathrm{Cos}[u2]}{\alpha\sqrt{\frac{(1+u1\,\alpha)^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\alpha^4}}}, -\frac{\mathrm{Sin}[u2]}{(1+u1\,\alpha)\sqrt{\frac{\alpha^2}{\alpha^2+2\,u1\,\alpha^3+u1^2\alpha^4}}}\right\}$$

- **Take limit as u1 approaches 0**

```
Limit[VinfPhys[u1, u2], u1 -> 0]
```

$\left\{\sqrt{\frac{1}{\alpha^2}}\ \alpha\,\mathrm{Cos}[u2],\ -\mathrm{Sin}[u2]\right\}$

- **Analytic solution to inviscid flow around a unit cylinder (physical components)**

$\mathrm{Vcyl}[u1\_,\ u2\_] = \left\{\left(1 - \frac{1}{(1+\alpha\,u1)^2}\right)\mathrm{Cos}[u2],\ -\left(1 + \frac{1}{(1+\alpha\,u1)^2}\right)\mathrm{Sin}[u2]\right\}$

$\left\{\left(1 - \frac{1}{(1+u1\,\alpha)^2}\right)\mathrm{Cos}[u2],\ \left(-1 - \frac{1}{(1+u1\,\alpha)^2}\right)\mathrm{Sin}[u2]\right\}$

$\mathrm{Pcyl}[u1\_,\ u2\_] = C - \frac{1}{2}\left(\frac{(1+\alpha\,u1)^4 - 2\,(1+\alpha\,u1)^2\,\mathrm{Cos}[2\,u2] + 1}{(1+\alpha\,u1)^4}\right)$

```
General::spell1 :
 Possible spelling error: new symbol name "Pcyl" is similar to existing symbol "Vcyl".
```

$C - \frac{1 + (1+u1\,\alpha)^4 - 2\,(1+u1\,\alpha)^2\,\mathrm{Cos}[2\,u2]}{2\,(1+u1\,\alpha)^4}$

- **Take limit as u1 approaches 0**

```
Limit[Vcyl[u1, u2], α -> 0]
```

```
{0, -2 Sin[u2]}
```

```
Simplify[Limit[Pcyl[u1, u2], α -> 0]]
```

```
-1 + C + Cos[2 u2]
```

■ Define the Navier Stokes Operator in General Coordinates

```
NavierStokes[v : {_, _}, p_, ν_] :=
     Advec[v]
   - ν LaplacianVector[v]
   + Sub2Sup[ Grad[p] ]
```

▪ Do the Stretching and Take Limits

▪ Define our starting equations with parameters

```
nsEq1[u1_, u2_, ν_] = NavierStokes[{v1[u1, u2], v2[u1, u2]}, p[u1, u2], ν]
```

$$\Big\{v2[u1,\ u2]\ \Big(-\frac{(1+u1\,\alpha)\ v2[u1,\ u2]}{\alpha}+v1^{(0,1)}[u1,\ u2]\Big)+\frac{p^{(1,0)}[u1,\ u2]}{\alpha^2}+v1[u1,\ u2]\ v1^{(1,0)}[u1,\ u2]-\frac{1}{\alpha^2\ (1+u1\,\alpha)^2}\ (\nu\ (-\alpha^2\,v1[u1,\ u2]-2\,\alpha\ (1+u1\,\alpha)\ v2^{(0,1)}[u1,\ u2]+\alpha^2\,v1^{(0,2)}[u1,\ u2]+v1^{(2,0)}[u1,\ u2]+2\,u1\,\alpha\,v1^{(2,0)}[u1,\ u2]+u1^2\,\alpha^2\,v1^{(2,0)}[u1,\ u2])),$$

$$\frac{p^{(0,1)}[u1,\ u2]}{(1+u1\,\alpha)^2}+v2[u1,\ u2]\ \Big(\frac{\alpha\,v1[u1,\ u2]}{1+u1\,\alpha}+v2^{(0,1)}[u1,\ u2]\Big)+v1[u1,\ u2]\ \Big(\frac{\alpha\,v2[u1,\ u2]}{1+u1\,\alpha}+v2^{(1,0)}[u1,\ u2]\Big)-\frac{1}{\alpha^2\ (1+u1\,\alpha)^3}\ (\nu\ (-\alpha^2\ (1+u1\,\alpha)\ v2[u1,\ u2]+2\,\alpha^3\,v1^{(0,1)}[u1,\ u2]+(1+u1\,\alpha)\ (\alpha^2\,v2^{(0,2)}[u1,\ u2]+2\,\alpha\,v2^{(1,0)}[u1,\ u2]+2\,u1\,\alpha^2\,v2^{(1,0)}[u1,\ u2]+v2^{(2,0)}[u1,\ u2]+2\,u1\,\alpha\,v2^{(2,0)}[u1,\ u2]+u1^2\,\alpha^2\,v2^{(2,0)}[u1,\ u2])))\Big\}$$

```
nsEq1[u1_, u2_, ν_] = Sup2Phys[nsEq1[u1, u2, ν]]
```

$$\Big\{\frac{1}{\sqrt{\frac{1}{\alpha^2}}}\ \Big(v2[u1,\ u2]\ \Big(-\frac{(1+u1\,\alpha)\ v2[u1,\ u2]}{\alpha}+v1^{(0,1)}[u1,\ u2]\Big)+\frac{p^{(1,0)}[u1,\ u2]}{\alpha^2}+v1[u1,\ u2]\ v1^{(1,0)}[u1,\ u2]-\frac{1}{\alpha^2\ (1+u1\,\alpha)^2}\ (\nu\ (-\alpha^2\,v1[u1,\ u2]-2\,\alpha\ (1+u1\,\alpha)\ v2^{(0,1)}[u1,\ u2]+\alpha^2\,v1^{(0,2)}[u1,\ u2]+v1^{(2,0)}[u1,\ u2]+2\,u1\,\alpha\,v1^{(2,0)}[u1,\ u2]+u1^2\,\alpha^2\,v1^{(2,0)}[u1,\ u2]))\Big),$$

$$\frac{1}{\sqrt{\frac{1}{(1+u1\,\alpha)^2}}}\ \Big(\frac{p^{(0,1)}[u1,\ u2]}{(1+u1\,\alpha)^2}+v2[u1,\ u2]\ \Big(\frac{\alpha\,v1[u1,\ u2]}{1+u1\,\alpha}+v2^{(0,1)}[u1,\ u2]\Big)+v1[u1,\ u2]\ \Big(\frac{\alpha\,v2[u1,\ u2]}{1+u1\,\alpha}+v2^{(1,0)}[u1,\ u2]\Big)-\frac{1}{\alpha^2\ (1+u1\,\alpha)^3}\ (\nu\ (-\alpha^2\ (1+u1\,\alpha)\ v2[u1,\ u2]+2\,\alpha^3\,v1^{(0,1)}[u1,\ u2]+(1+u1\,\alpha)\ (\alpha^2\,v2^{(0,2)}[u1,\ u2]+2\,\alpha\,v2^{(1,0)}[u1,\ u2]+2\,u1\,\alpha^2\,v2^{(1,0)}[u1,\ u2]+v2^{(2,0)}[u1,\ u2]+2\,u1\,\alpha\,v2^{(2,0)}[u1,\ u2]+u1^2\,\alpha^2\,v2^{(2,0)}[u1,\ u2])))\Big)\Big\}$$

- **Now multiply radial component by α**

```
nsEq2[u1_, u2_, ν_] = {{α, 0}, {0, 1}}.nsEq1[u1, u2, ν]
```

$$\left\{\frac{1}{\sqrt{\frac{1}{\alpha^2}}}\left(\alpha\left(v2[u1,u2]\left(-\frac{(1+u1\,\alpha)\,v2[u1,u2]}{\alpha}+v1^{(0,1)}[u1,u2]\right)+\frac{p^{(1,0)}[u1,u2]}{\alpha^2}+v1[u1,u2]\,v1^{(1,0)}[u1,u2]-\frac{1}{\alpha^2\,(1+u1\,\alpha)^2}\left(\nu\left(-\alpha^2\,v1[u1,u2]-2\,\alpha\,(1+u1\,\alpha)\,v2^{(0,1)}[u1,u2]+\alpha^2\,v1^{(0,2)}[u1,u2]+v1^{(2,0)}[u1,u2]+2\,u1\,\alpha\,v1^{(2,0)}[u1,u2]+u1^2\,\alpha^2\,v1^{(2,0)}[u1,u2]\right)\right)\right)\right),\right.$$
$$\left.\frac{1}{\sqrt{\frac{1}{(1+u1\,\alpha)^2}}}\left(\frac{p^{(0,1)}[u1,u2]}{(1+u1\,\alpha)^2}+v2[u1,u2]\left(\frac{\alpha\,v1[u1,u2]}{1+u1\,\alpha}+v2^{(0,1)}[u1,u2]\right)+v1[u1,u2]\left(\frac{\alpha\,v2[u1,u2]}{1+u1\,\alpha}+v2^{(1,0)}[u1,u2]\right)-\frac{1}{\alpha^2\,(1+u1\,\alpha)^3}\left(\nu\left(-\alpha^2\,(1+u1\,\alpha)\,v2[u1,u2]+2\,\alpha^3\,v1^{(0,1)}[u1,u2]+(1+u1\,\alpha)\left(\alpha^2\,v2^{(0,2)}[u1,u2]+2\,\alpha\,v2^{(1,0)}[u1,u2]+2\,u1\,\alpha^2\,v2^{(1,0)}[u1,u2]+v2^{(2,0)}[u1,u2]+2\,u1\,\alpha\,v2^{(2,0)}[u1,u2]+u1^2\,\alpha^2\,v2^{(2,0)}[u1,u2]\right)\right)\right)\right)\right\}$$

- **Now assume that the stretching parameter $\alpha=\sqrt{\nu}$, or $\nu=\alpha^2$**

```
nsEq3[u1_, u2_] = nsEq2[u1, u2, α^2]
```

$$\left\{\frac{1}{\sqrt{\frac{1}{\alpha^2}}}\left(\alpha\left(v2[u1,u2]\left(-\frac{(1+u1\,\alpha)\,v2[u1,u2]}{\alpha}+v1^{(0,1)}[u1,u2]\right)+\frac{p^{(1,0)}[u1,u2]}{\alpha^2}+v1[u1,u2]\,v1^{(1,0)}[u1,u2]-\frac{1}{(1+u1\,\alpha)^2}\left(-\alpha^2\,v1[u1,u2]-2\,\alpha\,(1+u1\,\alpha)\,v2^{(0,1)}[u1,u2]+\alpha^2\,v1^{(0,2)}[u1,u2]+v1^{(2,0)}[u1,u2]+2\,u1\,\alpha\,v1^{(2,0)}[u1,u2]+u1^2\,\alpha^2\,v1^{(2,0)}[u1,u2]\right)\right)\right),\right.$$
$$\left.\frac{1}{\sqrt{\frac{1}{(1+u1\,\alpha)^2}}}\left(\frac{p^{(0,1)}[u1,u2]}{(1+u1\,\alpha)^2}+v2[u1,u2]\left(\frac{\alpha\,v1[u1,u2]}{1+u1\,\alpha}+v2^{(0,1)}[u1,u2]\right)+v1[u1,u2]\left(\frac{\alpha\,v2[u1,u2]}{1+u1\,\alpha}+v2^{(1,0)}[u1,u2]\right)-\frac{1}{(1+u1\,\alpha)^3}\left(-\alpha^2\,(1+u1\,\alpha)\,v2[u1,u2]+2\,\alpha^3\,v1^{(0,1)}[u1,u2]+(1+u1\,\alpha)\left(\alpha^2\,v2^{(0,2)}[u1,u2]+2\,\alpha\,v2^{(1,0)}[u1,u2]+2\,u1\,\alpha^2\,v2^{(1,0)}[u1,u2]+v2^{(2,0)}[u1,u2]+2\,u1\,\alpha\,v2^{(2,0)}[u1,u2]+u1^2\,\alpha^2\,v2^{(2,0)}[u1,u2]\right)\right)\right)\right\}$$

- **Take the limit as $\nu\longrightarrow 0$ (or $\alpha\longrightarrow 0$) of the NavierStokes and divergence equations yields**

```
Limit[nsEq3[u1, u2], α -> 0]
```

$$\{p^{(1,0)}[u1,u2],\ p^{(0,1)}[u1,u2]+v2[u1,u2]\,v2^{(0,1)}[u1,u2]+v1[u1,u2]\,v2^{(1,0)}[u1,u2]-v2^{(2,0)}[u1,u2]\}$$

```
Limit[Diverg[{v1[u1, u2], v2[u1, u2]}], α -> 0]

v2^(0,1)[u1, u2] + v1^(1,0)[u1, u2]
```

Thus, this implies the following governing equations for the boundary layer flow in the limit of small viscosity:

$$\frac{\partial V_1}{\partial \tau} + \frac{\partial p}{\partial \zeta} = 0 \;\; (= \nu^{1/2}\,\kappa(\theta)\,V_1^{\,2} \text{ in general case } - \text{ for this case } \kappa = 1)$$

$$\frac{\partial V_2}{\partial t} + V_1 \frac{\partial V_2}{\partial \zeta} + V_2 \frac{\partial V_2}{\partial \theta} - \frac{\partial^2 V_2}{\partial \zeta^2} + \frac{\partial p}{\partial \theta} = 0,$$

$$\frac{\partial V_1}{\partial \zeta} + \frac{\partial V_2}{\partial \theta} = 0,$$

where $\tau = t / \nu^{1/2}$. These have the same form as Prandtl limit equations for a flat plate. However, now the boundary conditions are different. The outer boundary conditions, that is for $\zeta \to \infty$, are found from the matching conditions to the outer layer solution,which is the inviscid solution around the cylinder. An analytic solution is known for irrotational flow which provides the following outer boundary conditions:

$$V_1(\zeta, \theta) \xrightarrow{\zeta \to \infty} 0, \;\; V_2(\zeta, \theta) \xrightarrow{\zeta \to \infty} -2 \sin\theta,$$

The inner boundary conditions are taken directly as the boundary conditions at the surface of the cylinder. Assuming no-slip boundary conditions and control of the normal velocity we have

$$V_1(0, \theta) = c(\tau, \theta), \qquad V_2(0, \theta) = 0,$$

$$p(\zeta,\theta) \xrightarrow{\zeta \to \infty} \cos 2\theta \;\text{ or }\; 2\cos^2\theta \;\text{ or }\; -2\sin^2\theta \text{ depending upon choice of C}$$

where $c(\tau,\theta)$ is the normal (control) velocity on the surface of the cylinder.

APPENDIX B: Source Listing

At the end of this report we have included some source code of the computer implementation described in Section 3. Printouts of the C++ header files of the classes described in the text are listed and numbered individually. Most of these are the headers of the base classes upon which the implementation is based. A list of the headers is provided below along with a brief description.

FemLib.h This is the definition file for the class library FemLib. It provides necessary macro definitions for creating static and dynamics object libraries. It defines the namespace *FemLib* to which all classes in FemLib belong. It also defines some convenience classes like *R2* and the exception class for the FemLib library, *TSiFemLibError.*

ExtDef.h Auxiliary definition file containing extended type definitions (**typedef**'s).

FeNode2D.h The *FeNode2D* (actually named *TSiFeNode2D*) class definition. This class is basically a place marker for finite element formulations. It contains attributes describing its various characteristics including its system index and position (which is not really needed). Notice also that *FeNode2D* maintains a list of *FeElem2D* objects to which it is connected. The is also an iterator class *TSiFeNode2DElemIterator* which is used for iterating through all of those objects.

FeElem2D.h This is the base class for all finite element types in FemLib. It is basically a container of *FeNode2D* objects where the dependent variables are defined (or sampled if you will). It also includes an array of vertices, which are *R2* objects, needed for the differential geometry calculations.

FeQuad2D.h This is an actual finite element type in FemLib and is derived directly from *FeElem2D*. It implements all the functionality of an arbitrary quadrilateral in the 2D plane.

FeComplex2D.h The class *FeComplex2D* is a container of finite elements (*FeElem2D* objects) and nodes (*FeNode2D* objects) and represents a complex of finite element, that is, a geometric structure composed of polytopes that fit together "nicely". It provides methods for adding together these objects, as well as combining other complexes, to form the complete object. The classes *FeDomain2D* and *FeBoundary2D* are derived from this class.

FeGrid2D.h Base class for all finite element grids. A grid is composed of an array of *FeDomain2D* objects and an array of *FeBoundary2D* objects.

FeBodyInts.h This is a standalone class which compute various integrations used in the finite element analysis. The integrations are computed for individual *FeElem2D* objects. Once these integrals are computed for each finite element they may be pieced together to form the finite element system matrices. Note that a variety of numeric quadrature options are possible.

Iterators.h This is a template file containing templates for three different types of iterator classes. One for arrays, one for lists, and one for maps. It is included because iterator classes are frequently used in the FemLib library.

```
/*==================*\
||                  ||
||  Module FemLib   ||
||                  ||
\*==================*/

/* FemLib.h
 *
 * Macro Definitions Module for Library FemLib
 *
 *
 *
 *                   Author        : Christopher K. Allen
 *                   Copyright     : August, 1997
 *                   Last Revised  : September, 1997
 *
 *
 *
 */


/*
 *   Sentry
 */

#ifndef   FemLib_DEFINITION
#define   FemLib_DEFINITION



/*
 *   Create Library Importing/Exporting Macros
 */

#if       defined (FemLib_EXPORT)
#  define      FemLibImpExp  __declspec( dllexport )
#  define      FemLibFunc    __declspec( dllexport )
#  define      FemLibClass   __declspec( dllexport )

#elif     defined (FemLib_IMPORT)
#  define      FemLibImpExp  __declspec( dllimport )
#  define      FemLibClass   __declspec( dllimport )
#  if defined(__FLAT__)
#    define      FemLibFunc  __declspec( dllimport )
#  else
#    define      FemLibFunc
#  endif

#else
#  define      FemLibImpExp
#  define      FemLibFunc
#  define      FemLibClass

#endif



/*
 *   Include standard dependencies
 */

//
//  Avoid Name Collisions
//

#define NOMINMAX        // prevent min() and max() macros in WinDef.h
#define EXTDEF_NOINDEX  // prevent Index typedef in ExtDef.h


#ifndef   __AFX_H__
#include  <afx.h>
#endif

#ifndef   _INC_IOSTREAM
#include  <iostream.h>
```

```
#endif

#ifndef ExtDef_DEFINITION
#include  <ExtDef.h>
#endif

#ifndef MppExtDef_DEFINITION
#include  <MppExtDef.h>
#endif

#ifndef MathLib_DEFINITION
#include  <MathLib\MathLib.h>
#endif



/*
 *  Create the FemLib namespace and auxiliary classes
 */

namespace FemLib  {

  using namespace MathLib;



  /*
   *  Class TSiFemLibError
   */

  class FemLibClass TSiFemLibError {

  public:
      // User Interfaces
      //      Initialization
      TSiFemLibError();
      TSiFemLibError(Character* pErrString);
      ~TSiFemLibError();

      void        SetErrDescr(Character* pErrString);
      Character*  GetErrDescr();


  protected:
      // Attributes
      TextBuffer        bufDescr;         // error description

  };



};




#endif                    // FemLib_DEFINITION
```

```
/*=================*\
||                 ||
||  Header ExtDef  ||
||                 ||
\*=================*/


/*  ExtDef.h
 *
 *  Resource Module Containing Extended User Types
 *
 *  Contains definitions for type extension to standard C++ data
 *  types.  Provides convenient precision modifications and indicates
 *  context.
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : September, 1992
 *      Last Revised    : January, 1997
 *
 */


/*
 *  Declare Presence Identifer
 */

#ifndef   ExtDef_DEFINITION
#define   ExtDef_DEFINITION


//namespace ExtDef {

  // Alias basic data types
  typedef   char              Character;
  typedef   int               Integer;
  typedef   unsigned          Cardinal;
  typedef   double            Real;


  // New types
  typedef   void*                 Pointer;        // generic pointer
  typedef   Character*            STRING;         // beginning addr of string
  typedef   Character             TextBuffer[81]; // generic text buffer
  typedef enum  { False,True }    Boolean;        // simple Boolean, no operat's


  // Situation specific types
  typedef     char            Byte;      // basic memory unit
  typedef     long int        Filepos;   // file position type

#ifndef EXTDEF_NOINDEX
  typedef     unsigned        Index;     // indexing variable
#endif

//};


//  Windows (re)Definitions
typedef  int             BOOL;      // Boolean variable
typedef  unsigned short  WORD;      // machine word
typedef  unsigned long   DWORD;     // machine double word

// Boolean Variables
#define         FALSE          0
#define         TRUE           1


#endif
```

```
/*========================*\
||                        ||
||  Definition TSiFeNode2D ||
||                        ||
\*========================*/


/*  FeNode2D.h
 *
 *  Definition Module for Class TSiFeNode2D
 *
 *  The class TSiFeNode2D defines all the properties
 *  for a grid point in a finite element scheme.
 *
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : August, 1997 by Techno-Sciences, Inc.
 *      Last Modified   : August, 1997
 *
 */


/*
 *  Declare Presence Identifier
 */

#ifndef FeNode2D_DEFINITION
#define FeNode2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef     FemLib_DEFINITION
#include    <FemLib\FemLib.h>
#endif

#ifndef     Iterators_DEFINITION
#include    <FemLib\Iterators.h>
#endif

#ifndef   __AFXTEMPL_H__
#include  <afxtempl.h>
#endif


/*
 *  Declare in namespace FemLib
 */

namespace FemLib   {



  /*
   *    Forward Class Declaration
   */

  class FemLibClass   TSiFeElem2D;
  class FemLibClass   TSiFeBody2D;
  class FemLibClass   TSiFeFace2D;

  class FemLibClass   TSiFeDomain2D;
  class FemLibClass   TSiFeBoundary2D;

  class FemLibClass   TSiFeNode2DElemIterator;
  class FemLibClass   TSiFeNode2DFaceIterator;
  class FemLibClass   TSiFeNode2DBodyIterator;

  class FemLibClass   TSiFeNode2DDomIterator;
  class FemLibClass   TSiFeNode2DBndIterator;
```

```
/*
 *   Class TSiFeNode2D
 */

class FemLibClass TSiFeNode2D : public CObject {

    DECLARE_SERIAL( TSiFeNode2D )

public:

    friend TSiFeNode2DFaceIterator;     // give face iterator access
    friend TSiFeNode2DBodyIterator;     // give body iterator access
    friend TSiFeNode2DElemIterator;     // give elem iterator access
    friend TSiFeNode2DDomIterator;      // give dom iterator access
    friend TSiFeNode2DBndIterator;      // give bnd iterator access


    // New Type Definitions
    enum GridType    {
         gtGenericPt,
         gtInternalPt,
         gtBoundaryPt,
         gtCornerPt,
         nGridTypes,
         };

//    enum NodeType      {
//         ntGenericNode,
//         ntCentroidNode,
//         ntVertexNode,
//         ntEdgeNode,
//         nNodeTypes,
//         };

    enum BndCond   {
         bcNone,
         bcDirichlet,
         bcNeumann,
         bcNatural,
         bcControl,
         bcUser1,
         bcUser2,
         nBndConds,
         };


    typedef   CList<TSiFeElem2D*, TSiFeElem2D*>          ElemList;
    typedef   CList<TSiFeFace2D*, TSiFeFace2D*>          FaceList;
    typedef   CList<TSiFeBody2D*, TSiFeBody2D*>          BodyList;

    typedef   CList<TSiFeDomain2D*, TSiFeDomain2D*>      DomainList;
    typedef   CList<TSiFeBoundary2D*, TSiFeBoundary2D*>  BoundaryList;

    typedef   CArray<RealVector, RealVector&>            ValuesArray;


    // Globals
    static const STRING    sGridTypeNames[];      // string descriptions
//    static const STRING    sNodeTypeNames[];      // string descriptions
    static const STRING    sBndCondNames[];       // string descriptions

    static const Cardinal  nClassVersion;         // pstream class version


public:
    // User Functions
    //   Initialization
          TSiFeNode2D();                  // default constructor
          ~TSiFeNode2D();                 // destructor

    //   Definition
    void   SetGridType(GridType gt);      // grid domain location
```

```
//      void   SetNodeType(NodeType nt);      // set FE node type
     void   SetBndCond(BndCond bc);         // set boundary conditions

     void   SetCoord(R2& pt);               // set coordinate
     void   SetSysIndex(Cardinal i);        // set system node index

     BOOL   AttachFace(TSiFeFace2D*);       // attach face element to node
     BOOL   AttachBody(TSiFeBody2D*);       // attach body element to node

     BOOL   AttachDom(TSiFeDomain2D*);      // attach domain to node
     BOOL   AttachBnd(TSiFeBoundary2D*);    // attach boundary to node

     void   AddValues(RealVector& vec);     // add vector of node values

     //   Assignment
     TSiFeNode2D&     operator=(const TSiFeNode2D&);

     //   Data Query
     BndCond     GetBndCond()          { return bcNode; };
//      NodeType    GetNodeType()          { return ntNode; };
     GridType    GetGridType()         { return gtNode; };

     Cardinal    GetSysIndex()         { return iSysIndex; };
     R2          GetCoord()            { return ptCoord; };

     Cardinal    GetNumFaces()         { return lstFaces.GetCount(); };
     Cardinal    GetNumBodys()         { return lstBodys.GetCount(); };
     Cardinal    GetNumElems()         { return lstElems.GetCount(); };

     Cardinal    GetNumDoms()          { return lstDoms.GetCount(); };
     Cardinal    GetNumBnds()          { return lstBnds.GetCount(); };

     Cardinal    GetNumValues()        { return arrValues.GetSize(); };

     RealVector& GetValues(Cardinal i);

     //   Streams Support
     virtual void   StreamDump(ostream&);      // textual context dump
     virtual void   Serialize(CArchive& ar);   // pstream support


  protected:
     // Local Data
     GridType       gtNode;              // node type in domain
//      NodeType       ntNode;              // type of finite element node
     BndCond        bcNode;              // boundary conditions for node

     Cardinal       iSysIndex;           // index into system
     R2             ptCoord;             // coordinates in R2

     FaceList       lstFaces;            // list of attached face elements
     BodyList       lstBodys;            // list of attached body elements
     ElemList       lstElems;            // master list of attached elements

     BoundaryList   lstBnds;             // list of attached boundaries
     DomainList     lstDoms;             // list of attached domains

     ValuesArray    arrValues;           // value set for node

  protected:
     // Local Internal Functions
     //   Internal Support
     void   Initialize();                   // initialize data

     BOOL   AttachElem(TSiFeElem2D*);       // attach as general element
  };



  /*
   *  Inline Functions
   */

  //
```

```
//  Definition
//

inline void   TSiFeNode2D::SetGridType(GridType gt)
                { gtNode = gt; };

//  inline void   TSiFeNode2D::SetNodeType(NodeType nt)
//                  { ntNode = nt; };

inline void   TSiFeNode2D::SetBndCond(BndCond bc)
                { bcNode = bc; };

inline void   TSiFeNode2D::SetSysIndex(Cardinal i)
                { iSysIndex = i; };

inline void   TSiFeNode2D::SetCoord(R2& pt)
                { ptCoord = pt; };


//
//  Internal Support
//


//
//  Stream Support
//

inline  ostream&  operator<< (ostream& os, TSiFeNode2D& e)
      { e.StreamDump(os); return os; };




/*
 *   Class TSiFeNode2DFaceIterator
 */

class FemLibClass TSiFeNode2DFaceIterator :
  public TSiListIterator<TSiFeNode2D::FaceList, TSiFeFace2D>
{
public:
  TSiFeNode2DFaceIterator(TSiFeNode2D* pNode) :
    TSiListIterator<TSiFeNode2D::FaceList, TSiFeFace2D>(pNode->lstFaces) {};
};


/*
 *   Class TSiFeNode2DBodyIterator
 */

class FemLibClass TSiFeNode2DBodyIterator :
  public TSiListIterator<TSiFeNode2D::BodyList, TSiFeBody2D>
{
public:
  TSiFeNode2DBodyIterator(TSiFeNode2D* pNode) :
    TSiListIterator<TSiFeNode2D::BodyList, TSiFeBody2D>(pNode->lstBodys) {};
};


/*
 *   Class TSiFeNode2DElemIterator
 */

class FemLibClass TSiFeNode2DElemIterator :
  public TSiListIterator<TSiFeNode2D::ElemList, TSiFeElem2D>
{
public:
  TSiFeNode2DElemIterator(TSiFeNode2D* pNode) :
    TSiListIterator<TSiFeNode2D::ElemList, TSiFeElem2D>(pNode->lstElems) {};
};
```

```
/*
 *   Class TSiFeNode2DDomIterator
 */

class FemLibClass TSiFeNode2DDomIterator :
  public TSiListIterator<TSiFeNode2D::DomainList, TSiFeDomain2D>
{
public:
  TSiFeNode2DDomIterator(TSiFeNode2D* pNode) :
    TSiListIterator<TSiFeNode2D::DomainList, TSiFeDomain2D>(pNode->lstDoms) {};
};


/*
 *   Class TSiFeNode2DBndIterator
 */

class FemLibClass TSiFeNode2DBndIterator :
  public TSiListIterator<TSiFeNode2D::BoundaryList, TSiFeBoundary2D>
{
public:
  TSiFeNode2DBndIterator(TSiFeNode2D* pNode) :
    TSiListIterator<TSiFeNode2D::BoundaryList, TSiFeBoundary2D>(pNode->lstBnds) {};
};


};          // namespace FemLib


#endif    // FeNode2D_DEFINITION
```

```
/*===========================*\
||                           ||
||  Definition TSiFeElem2D   ||
||                           ||
\*===========================*/

/*  TSiFeElem2D.h
 *
 *  Definition Module for Class TSiFeElem2D
 *
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : August, 1997 by Techno-Sciences, Inc.
 *      Last Revised    : October, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef FeElem2D_DEFINITION
#define FeElem2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef FemLib_DEFINITION
#include  <FemLib\FemLib.h>
#endif

#ifndef FeNode2D_DEFINITION
#include  <FemLib\FeNode2D.h>
#endif

#ifndef    __AFXTEMPL_H__
#include  <afxtempl.h>
#endif


/*
 *  Declare in namespace
 */

namespace FemLib  {



  /*
   *  Forward Class Definition
   */

  class FemLibClass TSiFeElem2DNodeIterator;
  class FemLibClass TSiFeElem2DVertIterator;


  /*
   *  Class TSiFeElem2D
   */

  class  FemLibClass TSiFeElem2D : public CObject {

        DECLARE_SERIAL( TSiFeElem2D )

    public:
      // Give Iterators Access
      friend    TSiFeElem2DNodeIterator;
```

```
    friend    TSiFeElem2DVertIterator;


    // New Types
    enum  ElemType  {     // basic element types
          enmBasic,
          enmFace,
          enmBody,
          nElemTypes
          };

    typedef CArray<R2, R2&>                          FeVertArray;
    typedef CArray<TSiFeNode2D*, TSiFeNode2D*>   FeNodeArray;


    // Global Data
    static const STRING      sElemTypeNames[];        // string descriptors
    static const Cardinal    nClassVersion;           // pstream class version


  public:
    // User Interfaces
    //     Initialization
              TSiFeElem2D();
    virtual ~TSiFeElem2D();

    //    Definition
    void  SetSysIndex(Cardinal iSysIndex);            // set index into sys

    virtual void  SetNode(Cardinal i, TSiFeNode2D* pNode);// set node at index
    virtual void  SetVert(Cardinal iVertex, R2& pt);      // set vertex coord

    //    Data Query
    Cardinal   GetNumVerts()        { return nVerts; };
    Cardinal   GetNumNodes()        { return nNodes; };
    Cardinal   GetSysIndex()        { return iSysIndex; };

    TSiFeNode2D*  GetNode(Cardinal i);         // get ith node
    R2&           GetVert(Cardinal i);         // get ith vertex

    //    Event Hooks
    virtual void  StreamDump(ostream& os);    // textual content strmdump
    virtual void  Serialize(CArchive& ar);    // persistent strms support


protected:
    // Attributes
    ElemType        enmElemType;          // the basic element type

    Cardinal        nVerts;               // number of element vertices
    Cardinal        nNodes;               // number of nodes on element

    Cardinal        iSysIndex;            // index into FEM system

    FeVertArray     arrVerts;             // the element vertices
    FeNodeArray     arrNodes;             // nodes of the element


protected:
    // Internal Support
    void  Initialize();                   // initialize parameters

    //  For derived class
    void  SetElemType(ElemType enm);      // set basic element type
    void  SetNumNodes(Cardinal n);        // set number of nodes for element
    void  SetNumVertices(Cardinal n);     // set number of vertices for elem

};


/*
 *    Inline Functions
 */
```

```
    //
    //    Definition
    //

    inline void    TSiFeElem2D::SetSysIndex(Cardinal i)
                         { iSysIndex = i; };


    //
    //    Internal Support
    //

    inline void    TSiFeElem2D::SetElemType(ElemType enm)
                         { enmElemType = enm; };


    //
    //    Streams Support
    //

    inline ostream& operator<<(ostream& os, TSiFeElem2D& fe)
      {  fe.StreamDump(os);  return os;  };



    /*
     *    Iterator Classes
     */

    class FemLibClass TSiFeElem2DNodeIterator :
      public TSiArrayIterator<TSiFeElem2D::FeNodeArray, TSiFeNode2D>
    {
    public:
      TSiFeElem2DNodeIterator(TSiFeElem2D* pElem) :
        TSiArrayIterator<TSiFeElem2D::FeNodeArray, TSiFeNode2D>(pElem->arrNodes)
        {};

    };


    class FemLibClass TSiFeElem2DVertIterator :
      public TSiArrayIterator<TSiFeElem2D::FeVertArray, R2>
    {
    public:
      TSiFeElem2DVertIterator(TSiFeElem2D* pElem) :
        TSiArrayIterator<TSiFeElem2D::FeVertArray, R2>(pElem->arrVerts)
        {};

    };



};      // end namespace


#endif      // FiniteElement_DEFINITION
```

```
/*===========================*\
||                           ||
||  Definition TSiFeQuad2D   ||
||                           ||
\*===========================*/

/*  TSiFeQuad2D.h
 *
 *  Definition Module for Class TSiFeQuad2D
 *
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : August, 1997 by Techno-Sciences, Inc.
 *      Last Revised    : August, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef FeQuad2D_DEFINITION
#define FeQuad2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef   FeBody2D_DEFINITION
#include  <FemLib\FeBody2D.h>
#endif



/*
 *  Declare in namespace
 */


namespace FemLib  {

//  using namespace ExtDef;
//  using namespace MppExtDef;



  /*
   *  Forward Class Definition
   */

  class FemLibClass   TSiFeFace2D;



  /*
   *  Class TSiFeQuad2D
   */

  class  FemLibClass TSiFeQuad2D : public TSiFeBody2D {

        DECLARE_SERIAL( TSiFeQuad2D )

  public:
      // New Types
      typedef   TSiFeBody2D::Interpolation    Interpolation;

      // Global Data
      static const Cardinal   nClassVersion;         // pstream class version


  public:
      // User Interfaces
      //     Initialization
```

```
    TSiFeQuad2D();
    ~TSiFeQuad2D();

    virtual TSiFeFace2D* MakeFace(Cardinal i);          // make ith body face
    virtual void         MakeAllFaces();                // make all body faces

    //    FEM Requirements
    virtual void  SetInterpolation(Interpolation intp);

    virtual Real  ShapeFunc(Cardinal iNode, R2& ptLoc); // function value
    virtual R2    ShapeGrad(Cardinal iNode, R2& ptLoc); // function deriv

    virtual R2    GetCoordValue(R2& ptLoc);   // get coord from local coord
    virtual R2x2  GetCoordDeriv(R2& ptLoc);   // get coord derivative
    virtual Real  GetCoordJacob(R2& ptLoc);   // get coord xform Jacobian

    virtual BOOL  Membership(R2& ptGbl);      // pt in element?

    //    Event Hooks
    virtual void  StreamDump(ostream& os);
    virtual void  Serialize(CArchive& ar);


protected:
    // Attributes


protected:
    // Internal Support
    void  Initialize();

    //  Shape functions for interpolation scheme
    Real  ShapeFuncConst(Cardinal iNode, Real r, Real s);
    Real  ShapeFuncBiLin(Cardinal iNode, Real r, Real s);
    Real  ShapeFuncSeren(Cardinal iNode, Real r, Real s);
    Real  ShapeFuncQuad(Cardinal iNode, Real r, Real s);

    R2    ShapeGradConst(Cardinal iNode, Real r, Real s);
    R2    ShapeGradBiLin(Cardinal iNode, Real r, Real s);
    R2    ShapeGradSeren(Cardinal iNode, Real r, Real s);
    R2    ShapeGradQuad(Cardinal iNode, Real r, Real s);

    //  Used by Membership()
    BOOL  InsideSegment(R2& ptGbl, R2& ptSegV1, R2& ptSegV2);
};


/*
 *    Inline Functions
 */


//
//    FEM Requirements
//


//
//  Coordinate Functions
//


//
//    Streams Support
//

inline ostream& operator<<(ostream& os, TSiFeQuad2D& fe)
  {  fe.StreamDump(os);  return os;  };



};       // end namespace
```

```
#endif      // FeQuad2D_DEFINITION
```

```
/*=============================*\
||                             ||
||  Definition TSiFeComplex2D  ||
||                             ||
\*=============================*/

/*  FeComplex2D.h
 *
 *  Definition Module for Class TSiFeComplex2D
 *
 *
 *    TSiFeComplex2D is the container class for a finite element system
 *    representation in 2 dimensions.  It contains TSiFeNode2D objects
 *    and TSiFeElem2D Objects which define the system.
 *
 *    NOTE:
 *      A TSiFeComplex2D object will not allow duplicate nodes or elements.
 *      The members AddNode() and AddElem() will return FALSE if such an
 *      attempt is made.
 *
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : September, 1997 by Techno-Sciences, Inc.
 *      Last Revised    : October, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef FeComplex2D_DEFINITION
#define FeComplex2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef FeElem2D_DEFINITION
#include  <FemLib\FeElem2D.h>
#endif

#ifndef FeNode2D_DEFINITION
#include  <FemLib\FeNode2D.h>
#endif

#ifndef Iterators_DEFINITION
#include  <FemLib\Iterators.h>
#endif

#ifndef   __AFXTEMPL_H__
#include  <afxtempl.h>
#endif


/*
 *  Declare in namespace
 */


namespace FemLib  {


  /*
   *  Forward class definitions
   */

  class FemLibClass   TSiFeComplex2DNodeIterator;
  class FemLibClass   TSiFeComplex2DElemIterator;
```

```
/*
 *  Class TSiFeComplex2D
 */

class  FemLibClass TSiFeComplex2D : public CObject {

      DECLARE_SERIAL( TSiFeComplex2D )

  public:
    // Give the iterators access
    friend    TSiFeComplex2DNodeIterator;
    friend    TSiFeComplex2DElemIterator;


    // New Types
    typedef CArray<TSiFeNode2D*, TSiFeNode2D*>  FeNodeArray;
    typedef CArray<TSiFeElem2D*, TSiFeElem2D*>  FeElemArray;


    // Global Data
    static const Cardinal   nClassVersion;          // pstream class version


  public:
    // User Interfaces
    //      Initialization
              TSiFeComplex2D();
    virtual   ~TSiFeComplex2D();

    //    Definition
    void  SetHomogeneous(BOOL bHomog);      // all elements same size

    BOOL  AddNode(TSiFeNode2D* pNode);      // add node to grid
    BOOL  AddElem(TSiFeElem2D* pElem);      // add element to grid

    BOOL  AddComplex(TSiFeComplex2D* p);    // add objects of complex

    void  RemoveAll();                      // remove all objects
    void  DestroyAll();                     // destroy entire grid

    //    Data Query
    BOOL      GetHomogeneous()             { return bHomog; };

    Cardinal  GetNumElems()                { return nElems; };
    Cardinal  GetNumNodes()                { return nNodes; };

    TSiFeNode2D*  GetNode(Cardinal i);
    TSiFeElem2D*  GetElem(Cardinal i);

    Index         GetNodeSysIndex();

    //    Event Hooks
    virtual void  StreamDump(ostream& os);
    virtual void  Serialize(CArchive& ar);


protected:
    // Attributes
    BOOL            bHomog;             // is homogeneous grid ?

    Cardinal        nElems;             // number of element vertices
    Cardinal        nNodes;             // number of nodes on element

    FeElemArray     arrElems;           // array of elements
    FeNodeArray     arrNodes;           // array of nodes


protected:
    // Internal Support
    void  Initialize();

    BOOL  FindNode(TSiFeNode2D* p);     // true if node is in complex
    BOOL  FindElem(TSiFeElem2D* p);     // true if element is in complex
};
```

```
/*
 *    Inline Functions
 */

//
//    Definition
//

inline void   TSiFeComplex2D::SetHomogeneous(BOOL b)
                 { bHomog = b; };


//
//    Streams Support
//

inline ostream& operator<<(ostream& os, TSiFeComplex2D& fe)
  {  fe.StreamDump(os);  return os;  };


/*
 *  Class TSiFeComplex2DNodeIterator
 */

class  FemLibClass TSiFeComplex2DNodeIterator :
  public TSiArrayIterator<TSiFeComplex2D::FeNodeArray, TSiFeNode2D>
{
public:

  TSiFeComplex2DNodeIterator(TSiFeComplex2D* pGrid) :
    TSiArrayIterator<TSiFeComplex2D::FeNodeArray, TSiFeNode2D>(pGrid->arrNodes) {};

};


/*
 *  Class TSiFeComplex2DElemIterator
 */

class  FemLibClass TSiFeComplex2DElemIterator :
  public TSiArrayIterator<TSiFeComplex2D::FeElemArray, TSiFeElem2D>
{
public:

  TSiFeComplex2DElemIterator(TSiFeComplex2D* pGrid) :
    TSiArrayIterator<TSiFeComplex2D::FeElemArray, TSiFeElem2D>(pGrid->arrElems) {};

};



};       // end namespace


#endif       // NsSystem2D_DEFINITION
```

```
/*===========================*\
||                           ||
||  Definition TSiFeGrid2D   ||
||                           ||
\*===========================*/

/*  TSiFeGrid2D.h
 *
 *  Definition Module for Class TSiFeGrid2D
 *
 *
 *    TSiFeGrid2D is the container class for a finite element system
 *    representation in 2 dimensions.  It contains TSiFeNode2D objects
 *    and TSiFeElem2D Objects which define the system.
 *
 *    NOTE:
 *      A TSiFeGrid2D object OWNS all the objects it contains.  Therefore,
 *      once it goes out of scope all its objects are destroyed.
 *
 *
 *
 *      Author         : Christopher K. Allen
 *      Copyright      : September, 1997 by Techno-Sciences, Inc.
 *      Last Revised   : September, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef FeGrid2D_DEFINITION
#define FeGrid2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef FeBoundary2D_DEFINITION
#include  <FemLib\FeBoundary2D.h>
#endif

#ifndef FeObject2D_DEFINITION
#include  <FemLib\FeObject2D.h>
#endif

#ifndef FeDomain2D_DEFINITION
#include  <FemLib\FeDomain2D.h>
#endif

#ifndef Iterators_DEFINITION
#include  <FemLib\Iterators.h>
#endif

#ifndef   __AFXTEMPL_H__
#include  <afxtempl.h>
#endif


/*
 *  Declare in namespace
 */


namespace FemLib  {



  /*
   *  Forward class definitions
   */
```

```
class FemLibClass   TSiFeGrid2DNodeIterator;
class FemLibClass   TSiFeGrid2DElemIterator;


/*
 *  Class TSiFeGrid2D
 */

class  FemLibClass TSiFeGrid2D : public CObject {

      DECLARE_SERIAL( TSiFeGrid2D )

  public:
    // Give the iterators access
    friend    TSiFeGrid2DNodeIterator;
    friend    TSiFeGrid2DElemIterator;

    // New Types
    typedef CArray<TSiFeDomain2D*,   TSiFeDomain2D*>    DomArray;
    typedef CArray<TSiFeBoundary2D*, TSiFeBoundary2D*>  BndArray;
    typedef CArray<TSiFeObject2D*,   TSiFeObject2D*>    ObjArray;


    // Global Data
    static const Cardinal   nClassVersion;         // pstream class version


  public:
    // User Interfaces
    //     Initialization
              TSiFeGrid2D();
    virtual   ~TSiFeGrid2D();

    //     Definition
    void   AddDomain(TSiFeDomain2D* pDom);        // add a domain to grid
    void   AddBoundary(TSiFeBoundary2D* pBnd);  // add domain boundary to grid
    void   AddObject(TSiFeObject2D* pObj);       // add embedded object to grid

    void   RemoveDomain(Cardinal iDom);          // remove domain at index
    void   RemoveBoundary(Cardinal iBnd);        // remove boundary at index
    void   RemoveObject(Cardinal iObj);          // remove boundary at index

    void   DestroyAll();                         // destroy entire grid

    void   TagCornerNodes();                     // identify corner nodes


    //     Data Query
    Cardinal  GetNumNodes()             { return cpxMaster.GetNumNodes(); };
    Cardinal  GetNumElems()             { return cpxMaster.GetNumElems(); };

    Cardinal  GetNumDomains()           { return arrDoms.GetSize(); };
    Cardinal  GetNumBoundaries()        { return arrBnds.GetSize(); };
    Cardinal  GetNumObjects()           { return arrObjs.GetSize(); };

    TSiFeBoundary2D*  GetBoundary(Cardinal iBnd);
    TSiFeObject2D*    GetObject(Cardinal iBnd);
    TSiFeDomain2D*    GetDomain(Cardinal iDom = 0);
    TSiFeComplex2D&   GetMasterComplex();

    //     Event Hooks
    virtual void  StreamDump(ostream& os);
    virtual void  Serialize(CArchive& ar);


protected:
    // Attributes
    TSiFeComplex2D  cpxMaster;          // the master complex

    DomArray        arrDoms;            // array of domains
    BndArray        arrBnds;            // array of domain boundaries
    ObjArray        arrObjs;            // array of free boundaries


protected:
```

```
    // Internal Support
    void  Initialize();

};


/*
 *    Inline Functions
 */

//
//    Definition
//


//
//    Streams Support
//

inline ostream& operator<<(ostream& os, TSiFeGrid2D& fe)
  {  fe.StreamDump(os);  return os;  };



/*
 *  Class TSiFeGrid2dNodeIterator
 */

class  FemLibClass TSiFeGrid2DNodeIterator :
  public TSiFeComplex2DNodeIterator
{
public:

  TSiFeGrid2DNodeIterator(TSiFeGrid2D* pGrid) :
    TSiFeComplex2DNodeIterator( &(pGrid->cpxMaster) ) {};
};


/*
 *  Class TSiFeGrid2DElemIterator
 */

class  FemLibClass TSiFeGrid2DElemIterator :
  public TSiFeComplex2DElemIterator
{
public:

  TSiFeGrid2DElemIterator(TSiFeGrid2D* pGrid) :
    TSiFeComplex2DElemIterator( &(pGrid->cpxMaster) ) {};
};




};      // end namespace


#endif      // FeGrid2D_DEFINITION
```

```
/*===============================*\
||                               ||
||  Definition TSiFeBodyInts2D   ||
||                               ||
\*===============================*/

/*  FeBodyInts2D.h
 *
 *  Definition Module for Class TSiFeBodyInts2D
 *
 *
 *      Computes the shape function integrals in the finite element
 *      solution of the viscous, incompressible Navier-Stokes
 *      equations.
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : August, 1997 by Techno-Sciences, Inc.
 *      Last Revised    : October, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef FeBodyInts2D_DEFINITION
#define FeBodyInts2D_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef   FeBody2D_DEFINITION
#include  <FemLib\FeBody2D.h>
#endif

#ifndef   __AFXTEMPL_H__
#include  <afxtempl.h>
#endif



/*
 *  Declare in namespace
 */


namespace FemLib   {

//  using namespace ExtDef;
//  using namespace MppExtDef;



  /*
   *  Class TSiFeBodyInts2D
   */

  class  FemLibClass TSiFeBodyInts2D : public CObject {

        DECLARE_SERIAL( TSiFeBodyInts2D )

  public:
      // New Types
      enum  Quadrature  {
            enmNone,            // quadrature not set
            enmSimpson,         // exact for 3rd order poly
            enmBode,            // exact for 5th order poly
            enmGaussian3,       // good for 4th order poly
            enmGaussian4,       // good for 6th order poly
            enmGaussian5,       // good for 8th order poly
            enmExtSimpson,
```

```
        nQuadratures
        };

    typedef CArray<R2, R2&>      QuadPtArray;
    typedef CArray<Real, Real>   QuadWtArray;


public:
    // Global Data
    static const STRING     sQuadratureNames[];     // quadrature descr.'s

    static const Real       fSimpsonPts[];          // bode quad points
    static const Real       fSimpsonWts[];          // bode quad weights
    static const Real       fBodePts[];             // bode quad points
    static const Real       fBodeWts[];             // bode quad weights
    static const Real       fGaussianPts3[];        // gauss3 quad points
    static const Real       fGaussianWts3[];        // gauss3 quad weights
    static const Real       fGaussianPts4[];        // gauss4 quad points
    static const Real       fGaussianWts4[];        // gauss4 quad weights
    static const Real       fGaussianPts5[];        // gauss5 quad points
    static const Real       fGaussianWts5[];        // gauss5 quad weights

    static const Cardinal   nClassVersion;          // pstream class version


public:
    // User Interfaces
    //      Initialization
    TSiFeBodyInts2D();
    ~TSiFeBodyInts2D();

    //      FEM Configuration
    void    SetQuadrature(Quadrature qd);           // set num quadrature

    //      FEM Computations
    BOOL      CompBasisMatrix(TSiFeBody2D* pFe);    // comp stiffness matrix
    BOOL      CompGradMatrix(TSiFeBody2D* pFe);     // comp viscous mat
    BOOL      CompAdvecTensors(TSiFeBody2D* pFe);   // comp x advection ten
    BOOL      CompDivMatrices(TSiFeBody2D* pFeVel,  // comp x divergence mat
                              TSiFeBody2D* pFePres);

    //      Data Query
    RealMatrix&     GetBasisMatrix()      { return matBasis; };
    RealMatrix&     GetGradMatrix()       { return matGrad; };
    RealMatrix&     GetDivXMatrix()       { return matDivX; };
    RealMatrix&     GetDivYMatrix()       { return matDivY; };
    RealTensor&     GetAdvecXTensor()     { return tenAdvecX; };
    RealTensor&     GetAdvecYTensor()     { return tenAdvecY; };

    //      Event Hooks
    virtual void    StreamDump(ostream& os);
    virtual void    Serialize(CArchive& ar);


protected:
    // Attributes
    //      Configuration
    Quadrature          enmQuadrature;      // numeric quadrature type

    //      The integral matrices
    RealMatrix          matBasis;           // velocity basis matrix
    RealMatrix          matGrad;            // viscosity matrix
    RealMatrix          matDivX;            // divergence matrix in x dir
    RealMatrix          matDivY;            // divergence matrix in y dir
    RealTensor          tenAdvecX;          // advection tensor for x vel
    RealTensor          tenAdvecY;          // advection tensor for y vel

    //      Quadrature variables
    Cardinal            nQuadOrder;         // polynomial order of quadrature
    Cardinal            nQuadPts;           // number of quadrature pts total

    Real*               pQuadPts;           // array of standard quadrature pts
    Real*               pQuadWts;           // array of standard quadrature wts
    QuadPtArray         arrQuadPts;         // quadrature points
    QuadWtArray         arrQuadWts;         // quadrature weights
```

```
  protected:
      // Internal Support
      void     Initialize();

      void     InitQuadrature();          // init numeric quad

  };


  /*
   *    Inline Functions
   */


  //
  //    FEM Requirements
  //


  //
  //    Streams Support
  //

  inline ostream& operator<<(ostream& os, TSiFeBodyInts2D& fe)
    {  fe.StreamDump(os); return os;  };


};      // end namespace

#endif      // FiniteElement_DEFINITION
```

```
/*==========================*\
||                          ||
||  Definition TSiIterators ||
||                          ||
\*==========================*/

/*  Iterators.h
 *
 *  Definition Module for Class TSiIterators
 *
 *
 *
 *      Author          : Christopher K. Allen
 *      Copyright       : January, 1997 by Techno-Sciences, Inc.
 *      Last Revised    : April, 1997
 *
 */


/*
 *  Include Sentry
 */

#ifndef Iterators_DEFINITION
#define Iterators_DEFINITION


/*
 *  Include Interface Dependencies
 */

#ifndef __AFX_H__
#include  <afx.h>
#endif

#ifndef __AFXTEMPL_H__
#include  <afxtempl.h>
#endif



/*
 *  Declare in FemLib Namespace
 */

//namespace FemLib {

//  using namespace ExtDef;


  /*******************************************************
   *
   *  Class TSiArrayIterator
   *
   */

  template <class Array, class Object>
  class TSiArrayIterator  {

  public:
      // User Interfaces
      //      Initialization
      TSiArrayIterator(Array& rArray);
      ~TSiArrayIterator();

      void    Restart();                // restart the iterator

      //    Data Query
      Object*  GetCurrent()              { return pCurrObject; };

      Object*  operator++(int);          // return current Object and advance
               operator BOOL();          // still valid Objects's ?


  protected:
```

```
    // Attributes
    Array&        rArray;        // the attached map class
    Cardinal      pos;           // current position in map
    Object*       pCurrObject;   // the current argument
};


//
//  Member Functions
//

template <class Array, class Object>
TSiArrayIterator<Array, Object>::TSiArrayIterator(Array& _rArray) :
  rArray(_rArray)
{
  pos = 0;
  pCurrObject = NULL;
}


template <class Array, class Object>
TSiArrayIterator<Array, Object>::~TSiArrayIterator()
{}


template <class Array, class Object>
void  TSiArrayIterator<Array, Object>::Restart()
{
  pos = 0;
  pCurrObject = NULL;
}


template <class Array, class Object>
Object*  TSiArrayIterator<Array, Object>::operator++(int)
{
  if ( pos < (Cardinal)rArray.GetSize() )
    pCurrObject = (Object*)rArray.GetAt( pos++ );
  else
    pCurrObject = NULL;

  return pCurrObject;
}


template <class Array, class Object>
TSiArrayIterator<Array, Object>::operator BOOL()
{
  return ( pos < rArray.GetSize() );
}




/**********************************************************
 *
 *  Class TSiListIterator
 *
 */

template <class List, class Object>
class TSiListIterator  {

public:
    // User Interfaces
    //      Initialization
    TSiListIterator(List& _rList);
    ~TSiListIterator();

    void    Restart();               // restart the iterator

    //    Data Query
    Object*  GetCurrent()             { return pCurrObject; };

    Object*  operator++(int);         // return current Assoc and advance
```

```
            operator BOOL();        // still valid Assoc's ?


protected:
    // Attributes
    List&           rList;          // the attached map class
    POSITION        pos;            // current position in map
    Object*         pCurrObject;    // the current argument
};


//
//  Member Functions
//


template <class List, class Object>
TSiListIterator<List, Object>::TSiListIterator(List& _rList) :
  rList(_rList)
{
  pos = rList.GetHeadPosition();
  pCurrObject = NULL;
};


template <class List, class Object>
TSiListIterator<List, Object>::~TSiListIterator()
{};


template <class List, class Object>
void  TSiListIterator<List, Object>::Restart()
{
  pos = rList.GetHeadPosition();
  pCurrObject = NULL;
};


template <class List, class Object>
Object*   TSiListIterator<List, Object>::operator++(int)
{
  if (pos)
    pCurrObject = (Object*)rList.GetNext( pos );
  else
    pCurrObject = NULL;

  return pCurrObject;
};


template <class List, class Object>
TSiListIterator<List, Object>::operator BOOL()
{
  return (pos != NULL);
};



/********************************************************
 *
 *  Class TSiMapIterator
 *
 */

template <class Map, class Key, class Assoc, class Map_Assoc>
class TSiMapIterator  {

public:
    // User Interfaces
    //      Initialization
    TSiMapIterator(Map& rMap);
    ~TSiMapIterator();

    void    Restart();              // restart the iterator
```

```
    //    Data Query
    Assoc*  GetCurrent()            { return pCurrAssoc; };

    Assoc*  operator++(int);        // return current Assoc and advance
            operator BOOL();        // still valid Assoc's ?


protected:
    // Attributes
    Map&           rMap;       // the attached map class
    POSITION       pos;        // current position in map
    Assoc*         pCurrAssoc; // the current argument
};


//
//  Member Functions
//


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapIterator<Map, Key, Assoc, Map_Assoc>::TSiMapIterator(Map& _rMap) :
  rMap(_rMap)
{
  pos = rMap.GetStartPosition();
  pCurrAssoc = NULL;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapIterator<Map, Key, Assoc, Map_Assoc>::~TSiMapIterator()
{};


template <class Map, class Key, class Assoc, class Map_Assoc>
void  TSiMapIterator<Map, Key, Assoc, Map_Assoc>::Restart()
{
  pos        = rMap.GetStartPosition();
  pCurrAssoc = NULL;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
Assoc*   TSiMapIterator<Map, Key, Assoc, Map_Assoc>::operator++(int)
{
  Key          temp;
  Map_Assoc*   pMapAssoc;

  if (pos)
  {
    rMap.GetNextAssoc( pos, temp, pMapAssoc );
    pCurrAssoc = (Assoc*)pMapAssoc;
  }
  else
    pCurrAssoc = NULL;

  return pCurrAssoc;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapIterator<Map, Key, Assoc, Map_Assoc>::operator BOOL()
{
  return (pos != NULL);
};




/***********************************************************
 *
 *  Class TSiMapKeyIterator
 *
```

```
 */

template <class Map, class Key, class Assoc, class Map_Assoc>
class TSiMapKeyIterator  {

public:
    // User Interfaces
    //      Initialization
    TSiMapKeyIterator(Map& rMap);
    ~TSiMapKeyIterator();

    void    Restart();              // restart the iterator

    //    Data Query
    Key     GetCurrent()            { return key; };

    Key     operator++(int);        // return current Assoc and advance
            operator BOOL();        // still valid Assoc's ?


protected:
    // Attributes
    Map&          rMap;        // the attached map class
    POSITION      pos;         // current position in map
    Key           key;         // the current key
};


//
//  Member Functions
//


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapKeyIterator<Map, Key, Assoc, Map_Assoc>::TSiMapKeyIterator(Map& _rMap) :
  rMap(_rMap)
{
  pos = rMap.GetStartPosition();
//    key = NULL;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapKeyIterator<Map, Key, Assoc, Map_Assoc>::~TSiMapKeyIterator()
{};


template <class Map, class Key, class Assoc, class Map_Assoc>
void  TSiMapKeyIterator<Map, Key, Assoc, Map_Assoc>::Restart()
{
  pos        = rMap.GetStartPosition();
//    key = NULL;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
Key  TSiMapKeyIterator<Map, Key, Assoc, Map_Assoc>::operator++(int)
{
  Map_Assoc*  pMapAssoc;

  if (pos)
  {
    rMap.GetNextAssoc( pos, key, pMapAssoc );
  }
//    else
//      pCurrAssoc = NULL;

  return key;
};


template <class Map, class Key, class Assoc, class Map_Assoc>
TSiMapKeyIterator<Map, Key, Assoc, Map_Assoc>::operator BOOL()
{
  return (pos != NULL);
};
```

```
//};          // namespace FemLib

#endif       // Iterators_DEFINITION
```

Techno-Sciences, Inc.

10001 Derekwood Lane
Suite 204
Lanham, Maryland 20706
(301)577-6000

Control of Viscous Fluid Flow using Modal Analysis

Addendum to
Contract F49620-C-0020
Final Technical Report

March 27, 1998

Dr. Christopher K. Allen

Control of Viscous Fluid Flow using Modal Analysis

Christopher K. Allen

1 Introduction

In this short note we present a possible technique for designing feedback control systems for viscous fluid flow. The technique is based on the analysis of spatial modes generated by objects in the fluid domain and uses the group properties of the trigonometric expansion functions. We start with the two-dimensional vorticity transport equations and expand the field variables in terms of trigonometric exponentials. The time dependent equations for the expansion coefficients are then derived from the governing equations. The structure of these equations leads to a strategy for controlling the fluid motion.

2 The Vorticity Transport Equations

From the two-dimensional, incompressible Navier-Stokes equations we may derive the vorticity transport equations. These equations describe the time evolution and spatial distribution of the fluid's vorticity ω. This behavior is coupled to that of the fluid's stream function ψ as shown below.

$$\begin{aligned} \dot{\omega} - \nu\nabla^2\omega + J(\omega,\psi) &= 0 \\ -\nabla^2\psi &= \omega \end{aligned} \qquad (1)$$

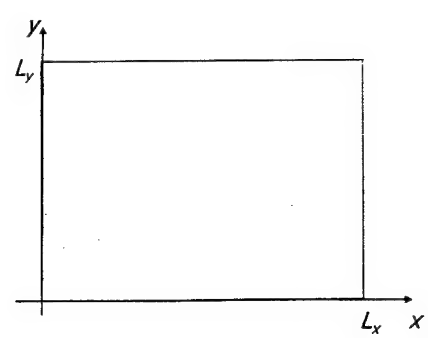
where the over dot indicates time differentiation, ν is the kinematic viscosity of the fluid, and $J(\omega,\psi)$ is the Jacobian determinant of ω and ψ (i.e., $J(\omega,\psi)=\partial(\omega,\psi)/\partial(x,y)$). The scalar functions ω and ψ are related to the vector valued velocity field $\boldsymbol{v}$ by the following:

$$\omega = \mathbf{a}_z \cdot \nabla \times \vec{v}, \qquad \vec{v} = \nabla \times (\mathbf{a}_z \psi). \quad (2)$$

Therefore, we can recover the fluid's velocity field from the above once system (1) is solved.

2.1 The Basis Functions

Consider the region shown in Figure 1 depicting a rectangular fluid domain in cartesian coordinates (x,y). The dimensions are L_x and L_y,



• Figure 1: Fluid domain

respectively. We let ω and ψ be elements of the function space spanned by the set of basis functions

$$\left\{e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}}\right\}_{\mathbf{n}=-\infty}^{\mathbf{n}=+\infty} \qquad (3)$$

where

$$\begin{aligned} \mathbf{n} &= (n_1, n_2) \in \mathbf{Z}\times\mathbf{Z} \\ \mathbf{x} &= (x_1, x_2) = (x, y) \in \mathbf{R}^2 \\ \mathbf{k}_\mathbf{n} &= (k_{n_1}, k_{n_2}) = \left(n_1\frac{2\pi}{L_x}, n_2\frac{2\pi}{L_y}\right) \end{aligned} \qquad (4)$$

Remarks:

1) The basis functions $e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}}$ are the eigenfunctions of the Laplacian operator ∇^2 with eigenvalues $-|\mathbf{k}_\mathbf{n}|^2 = -\left(k_{n_1}^2 + k_{n_2}^2\right)$. This makes inversion of the Laplacian operator particularly easy.

2) The basis $\left\{e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}}\right\}$ also forms a group under multiplication which is isomorphic to the group $\mathbf{Z}\oplus\mathbf{Z}$ with the isomorphism

$$e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}} \mapsto (n_1, n_2). \qquad (5)$$

3) The function space that $\left\{e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}}\right\}$ generates is then isomorphic to the group ring $\mathbf{C}(\mathbf{Z}\oplus\mathbf{Z})$. These properties are used when computing the Jacobian $J(\omega,\psi)$. The nonlinearities generated by the Jacobian term are represented by multiplication within the ring.

2.2 Computation of the Expansion Coefficients

The vorticity ω and stream function ψ may be expanded in terms of the above basis functions.

$$\omega(x,y;t) = \sum_{\mathbf{n}=(n_1,n_2)=-\infty}^{+\infty} \omega_\mathbf{n}(t)e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}} \quad \text{and} \quad \psi(x,y;t) = \sum_{\mathbf{n}=(n_1,n_2)=-\infty}^{+\infty} \psi_\mathbf{n}(t)e^{i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}} \qquad (6)$$

where

$$\begin{aligned} \omega_\mathbf{n}(t) &= \frac{1}{\sqrt{L_x L_y}} \iint_{[0,L_x]\times[0,L_y]} \omega(x,y;t), e^{-i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}} dxdy, \\ \psi_\mathbf{n}(t) &= \frac{1}{\sqrt{L_x L_y}} \iint_{[0,L_x]\times[0,L_y]} \psi(x,y;t), e^{-i\mathbf{k}_\mathbf{n}\cdot\mathbf{x}} dxdy. \end{aligned} \qquad (7)$$

The spatial variations are represented by the basis functions while the time variation is contained in the expansion coefficients $\omega_n(t)$ and $\psi_n(t)$.

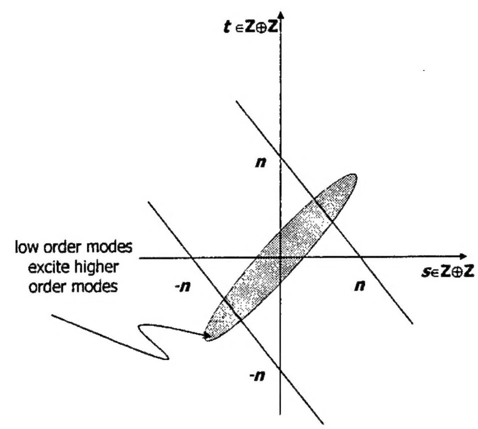
We insert these expansions into the vorticity transport equations to get first-order differential equations for the expansion coefficients. Note that from the second of Eqs. (1) we may express the ψ_n in terms of the ω_n by the orthogonality of the basis functions $\{e^{i\mathbf{k}_n \cdot \mathbf{x}}\}$. We have

$$\psi_n = \frac{\omega_n}{k_{n_1}^2 + k_{n_2}^2} \qquad (8)$$

Using this fact we need only consider the vorticity coefficients. Substituting the expansions (6) into (1) and using relation (8) yields (after carrying out the group multiplications)

$$\dot{\omega}_n + \nu(k_{n_1}^2 + k_{n_2}^2)\omega_n - \sum_{s+t=n} \frac{k_{s_1}k_{t_2} - k_{s_2}k_{t_1}}{k_{t_1}^2 + k_{t_2}^2}\omega_s\omega_t = 0 \qquad (9)$$

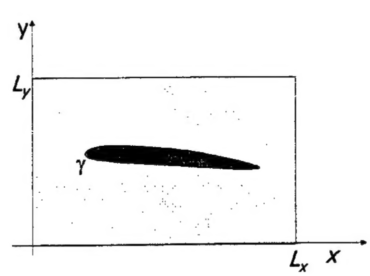
We have one such equation for every mode coefficient ω_n. (Recall that $\boldsymbol{n}$, $\boldsymbol{s}$, and $\boldsymbol{t}$ are all in the space $\mathbf{Z} \oplus \mathbf{Z}$.) The second term on the LHS of (9) represents the self-damping caused by the fluid viscosity while the third term (the summation) constitutes the dynamics of the coupling between modes in the fluid. Referring to Figure 2 we see schematically how low order modes can excite higher order modes through this process.



• Figure 2: Schematic of mode coupling

3 Controlling Fluid Flow

Now consider an object embedded in the fluid domain, say an airfoil. The situation is depicted in Figure 3. Also, assume that there is some sort of sensor/actuator assembly located on the airfoil that we are able to control. We wish to determine which spatial modes the airfoil can excite and, conversely, which modes the actuators can excite. Knowing this behavior we hope to design a control law which suppresses modes causing the most difficulty.



• Figure 3: Airfoil in fluid domain

3.1 Determining Mode Excitation

Assume that the boundary of the airfoil, γ, can be parameterized according to the following:

$$\begin{aligned} \gamma : [0,1] &\to [0, L_x] \times [0, L_y] \\ r &\mapsto (x(r), y(r)) \end{aligned} \qquad (10)$$

Now the set of restrictions $\left\{ e^{i\mathbf{k}_n \cdot \mathbf{x}} \big|_\gamma \right\}$ forms a basis for the space of complex analytic functions on the boundary γ, however, this basis is not orthogonal. We can use this condition to compute which modes the airfoil boundary will intrinsically excite. Let the function ω_0 defined on γ represent the no-slip boundary conditions on the airfoil (determination of ω_0 is not straight forward but assume for the sake of argument that it exists – boundary layer theory will probably be needed). Then we have the requirement

$$\omega(x, y; t) = \omega_0 \text{ on } \gamma. \qquad (11)$$

This requirement determines which modes are intrinsically excited by the airfoil. We may compute this excitement (approximately) by multiplying both sides of Eq. (11) by $e^{-i\mathbf{k}_m \cdot \mathbf{x}}$ then integrating over the boundary γ. Truncating at some $\mathbf{N}=(N_1,N_2)$ yields the matrix vector equation

$$\begin{pmatrix} \left\langle e^{i\mathbf{k}_1 \cdot \mathbf{x}}, e^{i\mathbf{k}_1 \cdot \mathbf{x}} \right\rangle_\gamma & \cdots & \left\langle e^{i\mathbf{k}_1 \cdot \mathbf{x}}, e^{i\mathbf{k}_N \cdot \mathbf{x}} \right\rangle_\gamma \\ \vdots & & \vdots \\ \left\langle e^{i\mathbf{k}_N \cdot \mathbf{x}}, e^{i\mathbf{k}_1 \cdot \mathbf{x}} \right\rangle_\gamma & \cdots & \left\langle e^{i\mathbf{k}_N \cdot \mathbf{x}}, e^{i\mathbf{k}_N \cdot \mathbf{x}} \right\rangle_\gamma \end{pmatrix} \begin{pmatrix} \omega_1 \\ \vdots \\ \omega_N \end{pmatrix} = \begin{pmatrix} \left\langle e^{i\mathbf{k}_1 \cdot \mathbf{x}}, \omega_0 \right\rangle_\gamma \\ \vdots \\ \left\langle e^{i\mathbf{k}_N \cdot \mathbf{x}}, \omega_0 \right\rangle_\gamma \end{pmatrix} \qquad (12)$$

where

$$\left\langle e^{i\mathbf{k}_m \cdot \mathbf{x}}, e^{i\mathbf{k}_n \cdot \mathbf{x}} \right\rangle_\gamma \equiv \int_0^1 e^{-i\mathbf{k}_m \cdot (x(r), y(r))} e^{i\mathbf{k}_n \cdot (x(r), y(r))} dr \qquad (13)$$

and

$$\left\langle e^{i\mathbf{k}_m \cdot \mathbf{x}}, \omega_0 \right\rangle_\gamma \equiv \int_0^1 \omega_0(r) e^{-i\mathbf{k}_m \cdot (x(r), y(r))} dr. \qquad (14)$$

Equation (12) can be inverted using standard techniques to determine the excited mode strengths.

Now let the function $\omega_a(t)$ defined on γ represent the action of the actuators on the airfoil. The boundary conditions for the actuators require that

$$\omega(x, y; t) = \omega_a(t) \text{ on } \gamma \qquad (15)$$

for all time t. With a similar argument as before, the modes and their strengths that the actuators excite at a particular time t is computed by the following equation:

$$
\begin{pmatrix}
\left\langle e^{i\mathbf{k}_1\cdot\mathbf{x}}, e^{i\mathbf{k}_1\cdot\mathbf{x}}\right\rangle_\gamma & \cdots & \left\langle e^{i\mathbf{k}_1\cdot\mathbf{x}}, e^{i\mathbf{k}_N\cdot\mathbf{x}}\right\rangle_\gamma \\
\vdots & & \vdots \\
\left\langle e^{i\mathbf{k}_N\cdot\mathbf{x}}, e^{i\mathbf{k}_1\cdot\mathbf{x}}\right\rangle_\gamma & \cdots & \left\langle e^{i\mathbf{k}_N\cdot\mathbf{x}}, e^{i\mathbf{k}_N\cdot\mathbf{x}}\right\rangle_\gamma
\end{pmatrix}
\begin{pmatrix} \omega_1 \\ \vdots \\ \omega_N \end{pmatrix}
=
\begin{pmatrix}
\left\langle e^{i\mathbf{k}_1\cdot\mathbf{x}}, \omega_a(t)\right\rangle_\gamma \\
\vdots \\
\left\langle e^{i\mathbf{k}_N\cdot\mathbf{x}}, \omega_a(t)\right\rangle_\gamma
\end{pmatrix}. \tag{16}
$$

Now that we know the modes that we have control over and their relative strengths, along with the modes that are excited naturally, we should be able to design a controller using Eq. (9) as a guide for the mode dynamics.

3.2 Other Application

It is possible to apply these same techniques to the Navier-Stokes equations directly in their primitive variable form. The results in that case will not, however, be as compact as they are here. Yet they may be more conducive to practical application. This may be especially true if we consider the restricted case of the time-varying boundary layer equations developed in the Phase I final report.